WEST Search History

Hide Items | Restore | Clear | Cancel

DATE: Sunday, March 21, 2004

Hide?	Set Name	Query	Hit Count
		DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=ADJ	
☐	L19	l18 and l9	1
☐	L18	packet near8 (protocol or processing) near8 (handler or manager or controller)	2307
☐	L17	packet near8 (protocol or processing)	38365
☐	L16	L15 and l3	0
☐	L15	((memory or queue or buffer) near8 (available near5 capacity)) same threshold same (allocate or allocating or allocated)	3
☐	L14	(memory or queue or buffer) near8 (available near5 capacity) near8 threshold near8 (allocate or allocating or allocated)	0
☐	L13	l9 and (DMA or (direct adj4 access))	9
☐	L12	L11 and l9	0
☐	L11	(queue or buffer) near8 (available near5 capacity) near8 threshold	21
☐	L10	(queue or buffer) near8 (available near5 capacity) near8 threshhold	0
☐	L9	20020225	19
☐	L8	((pipeline or pipelined or pipelining) near8 packet near8 (processing or process)) same (multi-processor or multi-processing or (multi adj3 (processor or processing)) or (multiple adj3 (processor or processing)) or thread or threading or multi-thread or (multiple adj3 thread))	25
☐	L7	(pipeline or pipelined or pipelining) near8 packet near8 (processing or process)	348
☐	L6	l3 and (DMA or (direct adj4 access))	18
☐	L5	l3 and DMA	10
☐	L4	L2 and ((queue or buffer) near8 capacity)	1
☐	L3	L2 and (queue or buffer)	34
☐	L2	20020225	36
☐	L1	((packet or packetized) near8 (pipeline or pipelined or pipelining)) same (multi-processor or multi-processing or (multi adj3 (processor or processing)) or (multiple adj3 (processor or processing)) or thread or threading or multi-thread or (multiple adj3 thread))	60

END OF SEARCH HISTORY

First Hit

Generate Collection

L4: Entry 1 of 1 File: PGPB Feb 27, 2003

DOCUMENT-IDENTIFIER: US 20030041163 A1
TITLE: Data processing architectures

Application Filing Date:
20020214

Detail Description Paragraph:
[0074] Part of a preferred feature of the present invention involves the application of multi-threading at a higher level to avoid this problem. The processing sequence is explicitly programmed in the multi-threaded fast-path code, and thread switches apply to a whole batch of packets being handled in parallel by a processor. While the global memory accesses for all packets are serialised and pipelined across the ClearConnect bus, the processor thread-switches and then executes another processing task that can operate on state held internally to the PEs. Exposing this mechanism to the programmer makes efficient operation of the system simple to achieve in high level software. The result is that when a processor accesses a global memory block, the memory accesses are efficiently overlapped with useful work on the PEs. Also, there is only a single memory latency cost for any number of accesses--the latency of all but the first access being overlapped with previous accesses, thus a high degree of immunity to memory latency can be achieved.

Detail Description Paragraph:
[0410] A line card solution at 10 Gbit/s could be readily built with current technology. Scaling in the above dimensions will soon yield a 40 Gbit/s solution, as outlined in the example given above- Only at the 100 Gbit/s point do scaling limits begin to be reached in terms of the number of processors sharing resources, or the efficiency of packet transport through the ClearConnect bus. The biggest problem at this performance is providing the capacity and bandwidth for the packet queue memory in the traffic management function.

First Hit

Generate Collection

L6: Entry 2 of 18 File: PGPB Feb 27, 2003

DOCUMENT-IDENTIFIER: US 20030041163 A1
TITLE: Data processing architectures

Application Filing Date:
20020214

Detail Description Paragraph:
[0074] Part of a preferred feature of the present invention involves the application of multi-threading at a higher level to avoid this problem. The processing sequence is explicitly programmed in the multi-threaded fast-path code, and thread switches apply to a whole batch of packets being handled in parallel by a processor. While the global memory accesses for all packets are serialised and pipelined across the ClearConnect bus, the processor thread-switches and then executes another processing task that can operate on state held internally to the PEs. Exposing this mechanism to the programmer makes efficient operation of the system simple to achieve in high level software. The result is that when a processor accesses a global memory block, the memory accesses are efficiently overlapped with useful work on the PEs. Also, there is only a single memory latency cost for any number of accesses--the latency of all but the first access being overlapped with previous accesses, thus a high degree of immunity to memory latency can be achieved.

Detail Description Paragraph:
[0086] The PEs themselves are optimised for the operations commonly performed on packet data, queue states and statistical information. They contain an 8-bit ALU datapath optimised for bit manipulation instructions, coupled to 32-bit wide 4 Kbyte local memory via a register file. PEs are arranged in a linear logical structure and each PE has direct communication to its two neighbours.

Detail Description Paragraph:
[0095] Any architecture that uses parallel hardware or memory structures to achieve the necessary bandwidth is subject to this problem. Many prior art solutions implement a reordering buffer somewhere in the system to restore the original packet order. Instead of adding this unwanted overhead to the system, the MTAP architecture of the present invention uses the storage of its PEs to perform this reordering as part of the fundamental Data Flow Processing solution.

Detail Description Paragraph:
[0128] Many of these problems are tackled with `DSP` solutions, which involve dedicated hardware or programmable processors optimised for efficient processing of signal data streams. A typical DSP system has a programmable processor containing several memory buffers. In order to maintain constant data streams in to and cut of the DSP processor, two DMA engines are employed, under control of the processor. The first DMA engine streams data from the system input, often via a FIFO buffer, to one half of a double-buffered section of memory in the processor. The processor has access to the data in the other half of the buffer in order to process it. The two halves of the input buffer can be swapped so that there is always memory for input data to arrive in, and there is always data for the processor to work on. The second DMA engine works in exactly the same way but transferring data from one half of a double-buffered memory in the processor to the system output, again sometimes

via a FIFO.

Detail Description Paragraph:
[0132] Streams of datagrams flow between processors. However, the transfer of these datagrams between processors cannot be direct. Processors must operate independently of each other, retrieving, processing and forwarding datagrams at their own rates. Furthermore, processor architectures and operating characteristics can result in these rates being non-uniform over time. The key to supporting processor independence is to insert memory based data buffers into every data path to de-couple the behaviours of each processor in the system from one another. Data buffers thus play a central role in enabling multiprocessor architectures to operate on continuous streams of data. They also facilitate system design by presenting standard interfaces which support well understood system operating principles (as described in the preceding section).

Detail Description Paragraph:
[0133] The use of data buffers for simple de-coupling in this way is not, by itself, inventive. The Inventiveness of this aspect of the invention lies in the way that data buffers car be designed to meet the unique requirements of SIMD processors whilst at the same time presenting robust and intuitive interfaces which are compatible with non-SIMD processors.

Detail Description Paragraph:
[0137] Consider a real time stream of data comprising datagrams of arbitrary and variable length. These datagrams are processed and forwarded individually. This is straight-forward for conventional MIMD architectures as a single processor can handle an entire datagram at a time. In SIMD architectures, however, the memory resource per processing element is limited and therefore fine grained distribution of datagrams across processors is necessary. Datagrams which are read from data buffer blocks by SIMD processors may thus be fragmented. Fragments, or `chunks` as they are referred to from hereon, are a fundamental characteristic of the proposed SIMD based system architecture for data stream processing according to this aspect of the invention.

Detail Description Paragraph:
[0143] A system in which all data flows in chunked form must be bounded. Boundary nodes must add chunk headers to datagrams entering the system, and remove headers from datagrams leaving the system. Although data buffers could be designed to straddle this boundary and perform chunking operations, it is preferable to define domain interface entities that encapsulate the chunking operations and any application specific functions. This then introduces two essential concepts:

Detail Description Paragraph:
[0148] Data buffers--targets which sit between processors and interfaces (or processors and processors)

Detail Description Paragraph:
[0151] The next step is to define how the flow of chunks through the system should be managed. Chunks provide a means of controlling the transfer of datagrams at the connection level--in a sense, a presentation format. However, the datagram itself remains the primary subject of interest. Processors act on whole datagrans. Data buffers forward datagrams.

Detail Description Paragraph:
[0152] Initiator to target transfer--When written to, data buffers consume datagrams in whatever chunked form they arrive in, de-chunk them and reconstruct the datagram.

Detail Description Paragraph:
[0154] Processors, MIMD and SIMD alike, can output datagrams to data buffers in any

chunked form as long as chunks are legitimate (eg. have appropriate headers and contain a single datagram or single fragment of a datagram). Chunk order of output datagrams should be preserved.

Detail Description Paragraph:
[0155] If multiple buffers are implemented in the data buffer then datagrams destined for different buffers may arrive in chunk interleaved format.

Detail Description Paragraph:
[0156] A datagram is only advertised to the data buffer output port (producer) when it is fully loaded into the buffer.

Detail Description Paragraph:
[0157] If the buffer overflows during the storage of a datagram then the remaining chunks are discarded and the partially stored datagram is purged from the buffer.

Detail Description Paragraph:
[0158] Target to initiator transfer--When read from, data buffers produce datagrams in a chunked form that is specified by the processor requesting the data. Two types of read operation are supported by data buffers for this purpose. The important concepts relating to Processors reading and writing data from/to data buffers are therefore that:

Detail Description Paragraph:
[0159] Processors read datagrams using a `batch read` mode. In a batch read, the processor issues a standardised request to the data buffer to send chunks. The request primarily is specifies the number of chunks and the maximum chunk size. The fact that the processor retains control of the transfer means that a system architecture is possible in which different, independent processors (SIMD and/or MIMD), in the same system could request data in different chunk sizes from the same data buffer.

Detail Description Paragraph:
[0160] Additionally, the processor can specify conditions which the data buffer must comply with. One such condition might specify whether datagrams may be split between batch read requests, i.e. whether the request should be terminated prematurely if a datagram cannot be transferred in full within the specified number of chunks. The addition of conditions make this a powerful feature. This conditional transfer by the data buffer under the instruction of the SIMD processor is a key feature of the SIMD data streaming architecture.

Detail Description Paragraph:
[0162] In the same way that processors exchange datagrams via shared data buffers, they can synchronise their activities and exchange control information via a shared resource called the Global Semaphore block. This block is used for high level software control to synchronise or force order between processor accesses to shared data buffers.

Detail Description Paragraph:
[0164] Control information can also pass between processors and data buffers. While it is the processor's responsibility to request and manage the transfer of datagrams to and from data buffers, it is necessary for the data buffers to inform the processors when they contain datagrams which the processor may wish to read. In a traditional architecture, interrupts may be used for this purpose. An interrupt from a data buffer could stimulate a SIMD processor to issue a batch read or a MIMD processor to issue a conventional read in order to retrieve a datagram from the buffer memory. Semaphores are proposed in place of interrupts as a more generic mechanism that can achieve the same result.

Detail Description Paragraph:

[0165] Direct memory access--transfer

Detail Description Paragraph:
[0166] In the batch read mode both chunk header and payload are delivered into the processor memory. This may not be desirable in all cases as more conventional processors may wish to read the control information first and then the payload (datagram) data. This is achievable as follows. When data buffers remove chunk headers and reconstruct datagrams, they store datagrams in a memory and retain information from the chunk headers separately. The datagram in addressable memory can thus be accessed directly by conventional memory read. Thus, the flexibility to be able to deliver datagrams to processors on request in subtly different formats is provided.

Detail Description Paragraph:
[0168] Data buffer functions in the SIMD data streaming architecture can accommodates this mode of operation thus enabling combined SIMD/MIMD systems.

Detail Description Paragraph:
[0170] Direct memory access--in-situ processing This can be regarded as an extension to DMA1. Data buffer memory could be used as a working memory on which the processor may operate--eg for large packets.

Detail Description Paragraph:
[0172] A specific implementation of the invention, in one particular context, has been outlined in the embodiment of the solution as described in section 1.2 of the co-pending application No GE 0102678.9, particularly in the design of data buffer blocks. Detailed descriptions of the data transfer modes, global application of the chunking scheme, and the architecture and operation of proposed data buffer blocks are found in chapter 3 of the co-pending application.

Detail Description Paragraph:
[0177] The most important component is the data buffer block. All manner of data buffer block types may be constructed from a set of basic modules. These modules collectively present a possibility for an inventive concept of a data buffer IP platform/toolkit for the design of data buffers to support SIMD based data streaming SoC design. The main components of the data buffer toolkit are:

Detail Description Paragraph:
[0180] The Buffer Manager

Detail Description Paragraph:
[0182] The toolkit components may be built into the following specific Data Buffer blacks:

Detail Description Paragraph:
[0183] Distributor (Single stream data buffer)--used for managing single data flows that are distributed to multiple SIMD processors arranged in parallel. Alternatively, the distributor may multiplex together multiple input streams or similarly act as a convergence point for a tributary flow joining the main flow.

Detail Description Paragraph:
[0184] Collector (Single stream data buffer)--used for managing single data flows which are collected from multiple SIMD processors arranged in parallel.

Detail Description Paragraph:
[0185] Datagram Exchange block (Multi-stream data buffer)--a software configurable datagram router which can be used to set up any required data stream path through the SoC without resource contention.

Detail Description Paragraph:

[0188] Control Plane Interface--The CPI block reuses the databuffer block toolkit IP to provide DMA and chunk/dechunk functions.

Detail Description Paragraph:
[0189] The combination of the component blocks (data buffer blocks, SIMD/MIMD processors, interfaces) and the system organisation concepts (chunking) gives the basis for the complete SIMD data flow processing architecture--the overall concept of using SIMD processor blocks to provide high power processing of continuous data streams.

Detail Description Paragraph:
[0278] A databuffer is a configurable block providing buffering of one or more queues. It operates on a consumer/producer model and allows for the co-ordination of packet flow through the system. Typical uses for a databuffer are in the distribution of batches of packets to processors in a cluster, or in the exchange of packets between the fast path data plane and the slow path control plane.

Detail Description Paragraph:
[0281] In accordance with the teaching of the present invention, a number of processors are combined with data buffer blocks to form a fast path Data Flow Processing architecture for data processing applications. Specific memory and/or hardware engines are added to provide dedicated hardware acceleration for certain application-specific processing functions. The ClearConnect bus is configured to provide the bandwidth needed to interconnect the different blocks in the fast path. Typically this will be explicitly expressed in the ClearConnect bus topology.

Detail Description Paragraph:
[0282] Slow path interactions, such as instruction fetch, table maintenance and packets to/from the control plane may use additional common buffers and interconnect. Interfaces are added at the system boundary for clean connection to physical line adapters, switch fabric adapters or other processing stages, including control plane processor. The total system is, in general, partitioned onto a number of these processing sub-systems arranged in a pipeline, as illustrated in FIG. 5. Stages may be identical or have different combinations of building blocks for different functions. One or more such stages may be integrated on a single chip. A number of degrees of hardware/software and headroom trade-offs can be made in partitioning the system.

Detail Description Paragraph:
[0301] Egress part numbers, queue identifiers and forwarding level of service characteristics are determined by performing a table lookup using a key constructed from various fields taken from the packet header.

Detail Description Paragraph:
[0308] As previously described, the table lockup operation that determines egress port numbers, queue identifiers and forwarding level of service characteristics is a function of networking layer QoS fields.

Detail Description Paragraph:
[0333] Referring back to FIG. 6, the principal components of the example network layer processing system are depicted. These items could represent the whole, or part of a single chip. The Network Input Port (NIP) and Network output Port (NOP) blocks convert between physical signalling protocols and the standard internal packet format used in the sub-system. The Distributor buffers incoming data from the NIP and transfers the data to the processor. The name "Distributor" is is derived from the block's capability to forward incoming data to multiple PEs. Conversely, the Collector buffers data sent by the PEs and multiplexes it into a single stream to the NOP.

Detail Description Paragraph:

[0335] A single processor consists of a number of processing elements and I/O engines with their associated controllers. FIG. 4 shows the major units within a processor. The control plane interface provides access to the fast path system and buffers packet data sent between the processor and the control plane.

Detail Description Paragraph:
[0339] The Distributor buffers incoming packet data. It forwards batches of packets to a processor when requested. The amount of packet data in a given batch of packets is (almost) constant while the number of packets in the batch varies in accordance with the distribution of packet size. Small packets are allocated to individual PEs within a processor. Large packets are allocated to a number of PEs sufficient for their storage.

Detail Description Paragraph:
[0341] Packets to be forwarded are sent from the processor to the Collector, which buffers and sends them to the NOP. The software running on the processors in a cluster enables them to co-ordinate with each other in a round-robin fashion to input, process and then output batches of packets. This preserves global packet order through the system.

Detail Description Paragraph:
[0347] Each PE accommodates multiple logical data buffers in its local memory in which packet data is stored. This allows new packet data to be loaded while the current packet data is being processed and old packet data is unloaded.

Detail Description Paragraph:
[0348] General-purpose semaphores are used to manage contention for shared resources such as PE memory buffers, access to the Distributor for read requests and access to the Collector for write requests.

Detail Description Paragraph:
[0363] Partitioning the 256 Processing Elements into four processors reduces the latency experienced by packets passing through the system. A batch of up to 64 packets is loaded into a processor, processed and then unloaded. A packet loaded and unloaded within a batch of 64 40 byte packets at 40 Gbit/s experiences a load/unload delay due to batching of approximately 0.5 .mu.s. Adding this to the processing time of almost 2 .mu.s gives a system processing latency of 2.5 .mu.s. Providing each Processing Element with 4 Kbytes of local memory enables a single processor to multi-buffer individual packets of up to 64 Kbytes in size.

Detail Description Paragraph:
[0410] A line card solution at 10 Gbit/s could be readily built with current technology. Scaling in the above dimensions will soon yield a 40 Gbit/s solution, as outlined in the example given above- Only at the 100 Gbit/s point do scaling limits begin to be reached in terms of the number of processors sharing resources, or the efficiency of packet transport through the ClearConnect bus. The biggest problem at this performance is providing the capacity and bandwidth for the packet queue memory in the traffic management function.

CLAIMS:

20. An architecture as claimed in claim 1, comprising a plurality of functional blocks chosen from: a SIMD processing element array, a data input device, a data output device, a hardware accelerator, a data packet buffer and a bus structure for connecting the functional blocks to one another.

First Hit

☐ Generate Collection

L6: Entry 3 of 18 File: PGPB Aug 29, 2002

DOCUMENT-IDENTIFIER: US 20020120828 A1
TITLE: Bit field manipulation

Abstract Paragraph:
A bit field direct manipulation device which processes data packets using a multi-threaded pipelined machine, wherein no instruction depends on a preceding instruction because each instruction in the pipeline is executed for a different thread. The multi-thread packet processor transfers a data packet from a flexible data input buffer to a packet task manager, dispatches the data packet from the packet task manager to a multi-threaded pipelined analysis machine, classifies the data packet in the analysis machine, modifies and forwards the data packet in a packet manipulator. The multi-thread packet processor includes an analysis machine having multiple pipelines, wherein one pipeline is dedicated to directly manipulating individual data bits of a bit field, a packet task manager, a packet manipulator, a global access bus including a master request bus and a slave request bus separated from each other and pipelined, an external memory engine, and a hash engine.

Application Filing Date:
20001222

Summary of Invention Paragraph:
[0007] Methods and apparatuses consistent with the principles of the present invention, as embodied and broadly described herein, provide for a method of directly manipulating a bit field without the use of separate insert or extract instructions. A bit field direct manipulation device processes data packets using a multi-threaded pipelined machine, wherein no instruction depends on a preceding instruction because each instruction in the pipeline is executed for a different thread. The multi-thread packet processor transfers a data packet from a flexible data input buffer to a packet task manager, dispatches the data packet from the packet task manager to a multi-threaded pipelined analysis machine, classifies the data packet in the analysis machine, modifies and forwards the data packet in a packet manipulator. The multi-thread packet processor includes an analysis machine having multiple pipelines, wherein one pipeline is dedicated to directly manipulating individual data bits of a bit field, a packet task manager, a packet manipulator, a global access bus including a master request bus and a slave request bus separated from each other and pipelined, an external memory engine, and a hash engine.

Detail Description Paragraph:
[0014] As shown in FIG. 1, an embodiment of the route switch packet architecture according to one aspect of the invention comprises Bi-directional Access Port (BAP) 10, Host Packet Injection (HPI) 14, Flexible Data Input Buffer (FDIB) 20, Test 28, Clock & PLLS 30, Analysis Machines (AMs) 42,56,70,84, Packet Task Manager (PTM) 98, Global Access Buses (GAB) 108,110,112,114,116,118, External Memory Engines (EME) 120,156, Internal Memory Engines (IME) 122,152, Packet Manipulator (PM) 126, Hash Engine (HE) 158, Centralized Look-Up Engine Interface (CIF) 160, Flexible Data Output Buffer (FDOB) 162, and Search/Results/Private 166,168. With the exception of Search/Results/Private 166,168, the combination of the above described elements may be considered a multi-thread packet processor.

Detail Description Paragraph:
[0027] In a packet processor, there is no explicit relationship from one packet to another packet except for the sequence of packets. The packets may be dispatched to multiple processing units or to multiple threads on a pipelined processing engine, as long as the packet sequence is maintained. Because of this, the multi-thread packet processor may be partitioned into multiple packet processing units, each being multi-threaded to keep all execution pipelines fully operating. Since this is a hardware partitioning, the packet sequencing is kept in hardware via PTM 98. As previously mentioned, the multi-thread packet processor may be designed for up to 250 MHz with 4 packet processing units providing 16.5 MPPS with 60 instructions used per packet forwarding decision.

Detail Description Paragraph:
[0028] Because the multi-thread packet processor processes the packets, it includes search capabilities. A common search metric used is the number of lookups per second the processor is capable of performing. The metric is typically bound, so that relative performance can be measured. Lookups using the radix-4 method can be effectively used in the routing of IP packets. The number of 24-bit radix-4 lookups for the multi-thread packet processor is a direct relation of the number of memory accesses EMEs 120,166 are able to do per second. (The lookup functionality is part of the External Memory Engine submodule.) The above-identified elements will be described in greater detail in the following sections.

Detail Description Paragraph:
[0030] BAP 10 may be designed for access by a general-purpose processor. All memory and register locations in the multi-thread processor address space are accessible from BAP 10. In an effort to make BAP 10 adaptable to future requirements, BAP 10 may be available to AMs 42,56,70,84 with the intention of reading status information from external peripheral devices. One application is the reading of external queue depths for use in implementing intelligent drop mechanisms. It is assumed that these algorithms only need to access the peripheral bus periodically. Thus, the interface can be shared with arbitrated host accesses. If host accesses are limited once a system is in a steady state, the multi-thread packet processor is capable of supporting accesses up to once per packet. At 16 million packets per second (MPPS), this equates to 16 million peripheral accesses per second. Thus, the multi-thread packet processor 250 MHz operation allows up to 15 cycles per access.

Detail Description Paragraph:
[0031] BAP 10 is configured as a shared multiplexed address and data bus that supports accesses to and from a generic host and peripheral devices. BAP 10 contains Global Registers 12, which include configuration and status registers that are global to the multi-thread packet processor. Registers that are specific to an element's function are contained in that element and accessible via one of the element's GAB interfaces. The operation of BAP 10 is controlled by BAP Global Registers 12. These registers include the source address, destination address, status register, interrupt vector, transfer size register, and several others. BAP's 10 interface to a host uses a chip select and ready control handshaking mechanism, allowing BAP 10 to interface with an external host operating at an unrelated asynchronous frequency. BAP 10 interfaces to all of the multi-thread packet processor's elements on each of the internal GABs 108,110,112,114,116,118. BAP 10 provides direct accesses to all internal memory and register locations for normal read and write operation types.

Detail Description Paragraph:
[0035] Flexible Data Input Buffer

Detail Description Paragraph:
[0037] FDIB 20 also contains the main packet buffering for the multi-thread packet processor. FDIB 20 includes four Packet Memories 26. Each of these memories may be

configured as a 512.times.128-bit dual port memory device that is segmented into 512 64-byte buffers. Each buffer has a page descriptor word contained in a separate 512.times.27 dual port memory. As pages fill, the descriptors are parsed and packet descriptors are generated with information including error-type (e.g., 3-bits), the length of the packet (e.g., 13-bits) as calculated by FDIB 20, and the master sequence number (e.g., 12-bits). Additionally stored are the receive port (e.g., 4-bits) and the address of the first page of the packet. All FDIB Packet Memories 26 and configuration registers are accessible by the host as well, with Packet Memories 26 being restricted to diagnostic mode access.

Detail Description Paragraph:
[0049] The state of each thread is independent from the state of all other threads. Threads and their register content are identified by a Thread Identification (TID) number. Status is provided to indicate which threads are active or inactive, enabled or disabled, etc. In addition to the AM integer pipeline that starts the execution of every AM instruction, each AM has access to several specialized co-processor units such as EMEs 120,266, HE 158, etc. The TID follows the instruction everywhere in the AM or co-processor pipelines. The TID is also the primary mechanism of control between all co-processing units, packet data interfaces, packet pre-classifiers, and the integer pipeline. For most of the interfaces, a TID Queue is used. Each TID queue is 16.times.4 bit FIFO that contains the thread identifications for some particular operation. Some of the TID queues have multiple write ports to allow new, continued, or co-processor return operations to be started simultaneously.

Detail Description Paragraph:
[0055] 3) Direct access of packet header memory.

Detail Description Paragraph:
[0071] Each AM includes packet pre-classification hardware. PTM 98 passes the length and address of the first buffer page of a packet to an AM thread. The next available thread takes the address and begins a fetch of the page into the Packet Header Memory contained in the AM. While the transfer is occurring over the AM's Packet Input GAB I/F, the pre-classification hardware snoops the data to classify the most basic known types. The hardware classification may be programmable and may be enabled or disabled. The concept of the hardware pre-classification is to aid the AM in a "fast dispatch" saving instructions for more critical processing. As such, pre-classification may be limited to well known protocols that make up 90-95% of the packet traffic. The pre-classification also aids in attempting to maintain line rate for packets smaller than 64-bytes. By pre-classifying some of the small packet types, less instructions can be used for these types, which in turn yields more processing power in the multi-thread packet processor and then the subsequent support of line rate for these as well.

Detail Description Paragraph:
[0083] Each of the AM threads has a context of its own registers and so on. The registers and packet memory are physically in a shared memory between the threads, but their direct access and use by a single thread makes them private. The private resources are as follows:

Detail Description Paragraph:
[0102] The direct memory access of AMs 42,56,70,84 however, may be limited to the Packet Header Memory (PHM) contained in each AM. The PHM is pre-loaded by an AM prior to starting a thread for packet processing, which is, coincidentally, when the AM performs the fast dispatch. The AM thread then has full access to the portions of the packet residing in the 64-byte PHM buffer. The 64-byte restriction is deemed sufficient as this will fit most known protocols with a reasonable descriptor attached. The AM thread also has the ability to go deeper in a packet with memory accesses from FDIB 20 or PHI to the PHM.

Detail Description Paragraph:
[0104] To combat this, the multi-thread packet processor allows direct manipulation of bit fields. The problem of bit field isolation, manipulation, and reintegration into the larger data item is handled by the underlying hardware rather than a sequence of instructions as would be done on a general purpose processor. The additional hardware increases the processing pipeline depth of each AM, but does not have a detrimental effect on the multi-thread packet processor throughput. For example, consider the problem of incrementing a 5-bit field within a word. The general-purpose processor generally needs to extract the field into a register, increment that register, and insert the field back. For AMs 42,56,70,84, this function is effected using a single instruction:

Detail Description Paragraph:
[0148] As shown in FIG. 3, the GAB is configured as a fully synchronous split operation protocol that is separated into two sections: Master Request Bus (MRB) 306,310 and Slave Result Bus (SRB) 308,312. Each operation starts with a master request and an MRB arbiter 302 grant. The MRB registers the operation to the slave devices. The operation is completed by a slave request and SRB arbiter 304 grant. The SRB registers the data back to the masters. The MRB and SRB are separated from each other and are pipelined. This allows multiple master requests to fill the pipelines of the slave devices, which are typically co-processing units, and then wait for the return data. Since the multi-thread packet processor master devices are typically multi-threaded, multiple pipelined requests may occur from any given master. Each slave and master has a ready signal to indicate that it is ready for the next operation. Masters assert their ready to the SRB arbiter and slaves assert their ready to the MRB arbiter. It is up to the designer of the master or slave device to insure that the ready signal is only asserted when the device is ready for the operations of which it is capable. For example, if a GAB device typically takes burst writes, then the ready signal should be asserted when there is enough room for a burst. Since the arbiter knows which device a master wants to target and has the slaves ready, an additional level of arbitration can implicitly be built in by not granting a master the GAB if the targeted slave is not ready. Similarly, the SRB can implicitly hold off a slave for return data if the master to return data to is not ready. This should not occur since the master had originally requested the operation.

Detail Description Paragraph:
[0149] The GAB Arbiter MUX (GAM) 300 submodule contains all the logic necessary for both the MRB and SRB: the arbiters, address/data/control MUXes, registers, and buffers.

Detail Description Paragraph:
[0154] 3. MRB Register Buffer

Detail Description Paragraph:
[0157] 6. SRB Register Buffer

Detail Description Paragraph:
[0158] The arbiters take the respective requests, readies, and the arbitration algorithm and grant a master (MRB) or slave (SRB) access to the split portion of the bus. The MRB MUX accepts select control from the MRB arbiter and multiplexes the various master signals to the MRB Register Buffer. All signals to the MRB from the masters should be registered outputs. The only incurred delay is the multiplexer structure. A single flip-flop for each data/address/control bit is provided in the MRB Register Buffer. Individual outputs with buffers are provided for each slave that needs a connection. The SRB Arbiter, SRB MUX and SRB Register Buffer work exactly the same as the MRB, except the operation types may be slightly different and the transfer is from one of the slave devices to one of the master devices.

Detail Description Paragraph:
[0186] Most operations on the MRB are single cycle since only a request needs to be transferred. The MRB registers and buffers the data to the slave being accessed. The MRB asserts the write signal to the slave, strobing in the request data. The slave performs the operation internal to its bounds. It then drives the appropriate return data, operation type, operation qualifier, master device/sub-device to return data to and address on its SRB GAB signals and asserts a request. The SRB arbiter eventually grants access to the slave. The grant is based on the other slave requests, the master ready, and the arbitration algorithm implemented. The SRB registers and buffers the operation return data to the master over the appropriate amount of cycles i.e., a burst read of 4 has 4 return data cycles at the master. Multiple slave destinations are allowed. Furthermore, the interleaving of slave return data on the SRB from two unique slaves back to one or more masters is also allowed and operates exactly the same. The MRB arbiter performs the operation and begins granting cycles based on the arbitration scheme and whether the requested slave is ready or not. A fair arbitration scheme is assumed, as well as the slave being ready. The 1 st cycle is granted to master 0, the 2nd to master 1, the 3rd to master 0, and the final cycle to master 1. Since no one is requesting, master 1 also gets the inadvertent grant which gets suppressed by the master asserting NOP, i.e. no write to the slave. The slave begins appropriate return operations to the master that requested it, by asserting its signals and requests. The latency of the response is dependent on the slave and the operation.

Detail Description Paragraph:
[0189] Packet Input GAB 106 provides an interface between AMs 42,56,70,84 and FDIB 20. An AM pulls the first buffer of the packet from FDIB 20 into the Packet Header Memory of the AM. During the initial transfer, as noted above, the AM Hardware Pre-Classifier snoops the packet and provides information to the AM thread. Subsequent accesses deeper into the packet are under full thread control through a predetermined instruction. Packet Input GAB 106 is one of the GABs in the multi-thread packet processor used for the flow of packet data. Packet Input GAB 106 transfers packet data from FDIB 20 to one of AMs 42,56,70,84. Typically, the transfer is the first page of a packet, but AMs 42,56,70,84 may access any number of words to the maximum burst in order to look deeper into a particular packet, if the protocol dictates. Packet Input GAB 106 has as its bus masters: all four AMs 42,56,70,84 and its slaves: FDIB 20 and HPI 14 submodules. The Packet Input GAB MRB uses TDMr arbitration. This allows fair access among AMs 42,56,70,84 while not starving BAP 10. Each AM is allocated one out of every four cycles. BAP 10 is given 4 out of 256 possible time slices of the TDM and is the default member of the round robin i.e., BAP 10 wins round robin only if no AM is requesting.

Detail Description Paragraph:
[0191] Control GAB 108 provides an interface between an AM and PTM 98. PTM 98 transfers packet length, input port, and the address of the first packet buffer in FDIB 20 of the packet. The AM is configured as both a master and a slave on Control GAB 108. The registers/memories of the AM are accessible via Control GAB 108 by BAP 10. The multi-thread packet processor uses Control GAB 108 or the flow of control information between various masters of the multi-thread packet processor. It is used primarily for packet notification, sequencing, and internal descriptor (message) passing. Control GAB 108 is also used for programming the instruction memories and configuration information into the AMs 42,56,70,84, PM 126 and PTM 98. Control GAB's 108 bus masters are: all AMs 42,56,70,84, PTM 98 and BAP 10 submodules. Control GAB's 108 slaves are: all AMs 42,56,70,84, PTM 98, PM 126, FDIB 20 and HPI submodules.

Detail Description Paragraph:
[0207] Each EME may access two separate 36-bit memory banks independently. Even parity is implemented for each byte during 32 and 64-bit accesses. An AM 40,56,70,84 may issue burst reads or writes of 8 over the GAB, while PM 126 may issue a burst read of 8 using a separate high-speed access port with a 64-bit wide

data bus. A 64-bit access from PM 126 is split across both banks, while a similar access from an AM 40,56,70,84 may cause the EME to double pump a single bank. 64-bit accesses from an AM and PM 126 may be word aligned. Each bank supports memory expansion for 4 SSRAMs, 256K in depth each. When memory is expanded, a register bit is set to remove a pipeline stage in the read and write paths. This means the issue and receive queues are "shortened" by one entry so there is an extra clock cycle available to perform external address decode and data MUXing. The data bus MUXing between memories may be accomplished with transceivers so that all connections are point-to-point to run the memories at 200 MHz. When the EME is used in the CLUE, the signals for the high-speed access port may be tied off and the related logic optimized out during synthesis.

Detail Description Paragraph:
[0208] As such, lookup performance may reach 25+ million lookups per second for a single memory bank. This is based on a 24-bit key for a IP-V4 lookup, requiring 7-8 memory cycles running at 200 MHz. Assuming the lookup tables are well distributed across memory banks, an EME could reach 50+ million lookups per second using both memory banks. Lookup performance is reduced when these operations have to compete with reads/writes from AMs 40,56,70,84 and PM 126. Each EME is pipelined to improve speed and mirror external memory. There are two pipelines, one for each bank that operate independently of each other. Hence, there are two separate arithmetic and logic units (ALUs), two write buffers, etc.

Detail Description Paragraph:
[0211] The high-speed access port (HSAP) controller contains asynchronous FIFOs and control logic to handle burst reads from PM 126. The HSAP controller increments the address for burst reads so they appear as single reads to the EME pipelines. Both pipelines operate on the read request simultaneously since a PM read is normally 64-bits wide and split across both banks. The two data streams are merged at the PM outbound FIFO. Parity is checked, if enabled. If a parity error is detected, a parity error signal is asserted to PM 126 and to BAP 10. The write buffer in each pipeline may not used for PM accesses because the software may force a write buffer flush before PM 126 accesses the data. During the final write from an AM, a field may specify a flush operation. When the buffer is flushed, this write may be acknowledged on the GAB slave return bus so the AM knows the data is in external memory. The AM may then launch a job packet to PM 126, which can then access the data from external memory. The control logic in the HSAP controller handshakes with the SSRAM state machines in each pipeline so that all PM requests can be serviced immediately. The HSAP controller also contains a request FIFO to absorb multiple read requests to remove any bandwidth penalty associated with handshaking across an asynchronous boundary.

Detail Description Paragraph:
[0212] The input and output FIFOs buffer data flow between the pipelines and the MUXs. Since lookups and filters can be forwarded from one bank to the other (depending on the contents of the bank forwarding registers), a lockout condition can occur where the output FIFOs for each bank are full and each input FIFO has a lookup that needs to continue in the "other" bank. This is controlled by the MUXs that do not allow more than 32 operations to be submitted across both pipelines. The input FIFOs are 32 deep so, regardless how the operations flow through the pipelines and FIFOs, all operations can be absorbed by either input FIFO during a stall condition (PM access) so all lockout scenarios are avoided. A 64-bit access is counted as two operations and burst accesses are handled similarly. The MUXs increment a counter whenever something is entered into either output FIFO and is decremented whenever something is taken from either input FIFO that is destined for the GAB.

Detail Description Paragraph:
[0213] The EME pipeline is a complex configuration that contains a Write Buffer, an ALU, and a Loopback FIFO. The EME directly controls external SSRAM, and services PM

requests. The write buffer consists of a 64-bit wide by 8-deep memory to store data along with a "parallel" set of flops that store a 20-bit address, a pair of valid (V) bits, and a pair of reserved (R) bits. The write buffer can behave as a cache since the address of all requests from the output FIFO are compared with the write buffer addresses. However, this is not the main purpose of the buffer because most addresses to memory have random behavior, minimizing the probability of a hit in the write buffer. The main goal of the write buffer is to reduce bus turnaround time penalties by writing the data as a burst during a flush sequence. Since the buffer may contain eight 64-bit entries, this could take up to 16 clock cycles. If there is a PM access during a flush, the SSRAM state machine stalls the write buffer flush, turns the bus around to read data for PM 126, then turns the bus around again to finish the write buffer flush.

Detail Description Paragraph:
[0214] A write buffer flush is triggered under the following circumstances: (1) a write is present in the Output FIFO and the write buffer is full; (2) the write buffer flush register bit is set; (3) a write is issued with bit 2 of a field set. When a write buffer flush is in progress, the write that is present in the Out FIFO is also sent to memory before the bus is turned around for reads. The write with flush option may be used when updating lookup tables and PM data structures. This is because addresses are compared at the Out FIFO and not at the Loopback FIFO or at the PM interface. Neglecting to flush the write buffer may cause PM 126 to retrieve "stale" data from external memory.

Detail Description Paragraph:
[0215] The ALU performs all the arithmetic functions for atomic and statistical adds, including the "stickiness" feature, as well as address calculation for lookups and filters. When there is a hit in the write buffer while an atomic or statistical add is issued from the Out FIFO, a memory cycle is wasted as the operation travels from the outbound pipeline to the inbound pipeline, dropping the read data from external memory to use the data in the buffer instead. A large MUX before the ALU controls data flow, selecting the most recent data during back-to-back atomic operations using the same address. Output and input delay cells are added to improve setup/hold times in the read/write paths to external memory. There is a 2-to-1 MUX to select data for memory writes, using a memory control signal from a register bit. This signal is low when late-write SSRAMs are used, so the data is driven one clock cycle after the address. If a different memory is used where data must be driven two cycles after (i.e., burst mode SSRAMs), a register bit can be set to flip the MUX to select data from the next stage in the pipeline. Similarly, a MUX using a memory expansion signal selects which address and associated tag information corresponds to the incoming SSRAM data. When expanded memory is used, the address is delayed a couple clock cycles to match the extra external delay where one additional clock cycle is allowed for external address decode and data MUXing, and a second extra cycle to register the read data externally before it is supplied to the multi-thread packet processor.

Detail Description Paragraph:
[0216] The pipeline flow is best explained by describing how basic operations travel through the various stages. Before discussing how reads are processed, it is important to understand how the SSRAM state machine controls the flow of data from the Out FIFO, the Loopback FIFO, the Write Buffer during flushes, and all PM requests. The SSRAM state machine can stall the outbound pipeline that is fed from the Out FIFO whenever there is a PM access. Conversely, the inbound pipeline that contains the ALU cannot be stalled. Whenever the outbound pipeline is stalled, all operations from the inbound pipeline are absorbed by the Loopback FIFO or Input FIFO. If a read is supplied by the Out FIFO, the SSRAM state machine allows it to enter the outbound pipeline if there are no PM accesses, the Loopback FIFO is empty, and no write buffer flush is in progress. If there is an address match with one of the entries in the write buffer, a tag bit is set along with a 4-bit address that corresponds to the physical address in the write buffer. The upper three

address bits are used to supply the address while the lowest bit selects the upper or lower 32-bit word when the read reaches the write buffer. For a 36 or 64-bit read, this lowest address bit is ignored. Once the read enters the outbound pipeline, the address is driven on the next cycle while the same address and related tag information travels in the pipeline.

Detail Description Paragraph:
[0217] Once the address/tag reaches the inbound pipe, the read data from the SSRAM arrives at the same cycle so they travel as a pair up the inbound pipe, destined for the PM, In FIFO, or Loopback FIFO. If the read is for a 64-bit access, the SSRAM state machine may issue the original word-aligned address during the first cycle, and then reissue the same address with bit 0 set to access the upper 32 bits of the 64-bit word. The state machine stalls the outbound pipeline during this second access. When the data arrives at the inbound pipeline, each 32-bit word is parity checked if parity checking is enabled. The two 32-bit words then enter a MUX to be merged as a single 64-bit word (single stage) before the data is issued to the ALU. If there were a hit in the write buffer, this data would have been MUXed instead, dropping the data from external memory. For a read, the ALU simply passes the data along to the In FIFO, destined for the GAB. A write operation travels down the outbound pipeline if there is room in the write buffer and the buffer is enabled. As always, the SSRAM state machine gives priority to entries in the Loopback FIFO so it must be empty before any operations are accepted from the Out FIFO. The write address may be stored in the write buffer and marked as reserved. If its a 64-bit write, two reserve bits may be set to indicate that the upper and lower 32 bits will be stored. The write may be issued, but tagged as a read in the pipeline so the SSRAM state machine does not turn the bus around. Once the address/tag reaches the inbound pipeline, the data is dropped because the tag information is still a write.

Detail Description Paragraph:
[0218] When the address/tag/write data reaches the ALU, the data is written to the write buffer and the entry is marked as valid (2 valid bits for a 64-bit write). At the same time, the write is returned to the In FIFO so a write acknowledge can be sent out the GAB. If parity is enabled and a write buffer flush occurs, all 32 and 64-bit write data is passed through the parity generation logic. A 64-bit write is stalled in the pipeline until the upper and lower 32-bit words are written to memory, using the MUX to steer the data to the final output flops. If a 64-bit write were destined for the write buffer instead, the parity generation logic is bypassed and a single read cycle is wasted as the data is dropped at in inbound pipeline. During a write buffer flush, none of the writes return to the GAB as write acknowledges since they were already sent when the write buffer was initially loaded.

Detail Description Paragraph:
[0219] Atomic and statistical adds operate in a similar manner to writes, reserving an address in the write buffer. If the buffer is full, it is flushed before the add is taken from the Out FIFO. The read data from external memory is parity check (32 or 64 bit access) and supplied to the MUX in the inbound pipeline. If the address in the write buffer is only reserved and not valid, the read data from memory is supplied to the ALU for the addition. In the case of back-to-back atomic operations to the same address, the most recent data is ahead of the ALU and not necessarily in the write buffer. In this case, the MUX selects the data after the ALU in one of the pipeline stages, instead of the data from external memory. Once an atomic operation passes through the ALU, the data is written to the write buffer and the result is also passed to the In FIFO to issue an acknowledge cycle on the GAB.

Detail Description Paragraph:
[0220] Lookups and filters enter the outbound pipeline the same way as reads, however, the address is not compared with the addresses in the write buffer. Lookups and filters are treated as reads in the outbound pipeline and the read data

arrives at the ALU in the inbound pipeline. The ALU calculates the lookup address if the continue bit is set and the new address and remainder of the key travel into the Loopback FIFO if the lookup needs to proceed in the same bank and same EME. The bank forwarding registers control the flow of lookups and filters. Later sections in this chapter provide a more thorough explanation of lookups and filters.

Detail Description Paragraph:
[0221] If a lookup/filter needs to continue in a different bank or EME, or the search has ended, the result is passed to the In FIFO. All lookups and filters appear at the Out FIFO, with a given address and a MSB aligned key. Lookups/filters that continue by entering the In FIFO have the next search address and the MSB aligned key. If it needs to go to the other bank, the MUX transfers the data to the other Out FIFO, otherwise the key is LSB aligned before the lookup/filter is sent out the GAB to another EME. in one implementation, the difference between MEMCLK and the return clock should not exceed 2nS since there is no logic between the first and second stage of flops. During 32 and 64-bit read operations, parity is verified if enabled by setting a bit in the configuration register. If a parity error occurs, a bit is set in the status register and a parity error signal is asserted to BAP 10. A 64-bit read double pumps a single memory and is always word aligned, meaning address bit zero is ignored. If the write buffer is enabled, the read address is compared with the addresses stored in the write buffer. Regardless of a hit or miss, the read travels down the outbound pipeline to initiate a SSRAM read. The read data is replaced with the data in the write buffer or from one of the feedback paths after the ALU. This decision is done by the MUX block one pipeline stage before the ALU. The read proceeds to the In FIFO, and then finally to the GAB using the device and sub-device information that travels with the read through the pipeline.

Detail Description Paragraph:
[0222] A read with clear tag information is treated as an atomic operation, except the original value is sent to the GAB while a value of all zeros is written to the write buffer. A register read never enters the pipeline since all registers are present in the MUX. Register reads are responded to immediately and do not follow the order of operations in the pipeline. They remain ordered compared to other register accesses and are never inhibited unless the GAB slave return bus is stalled. If the pipelines are disabled via a register bit, any operations destined for either pipeline are dropped so reads can pass through the master request GAB FIFO and complete without interruption.

Detail Description Paragraph:
[0223] Write operations have the same address mapping as reads, and address matching is applied if the write buffer is enabled. The write travels down the outbound pipeline as a read, and the read data is dropped in the inbound pipeline. The write data is written to the reserved location in the write buffer after passing through the ALU. Parity is not generated until the write buffer is flushed since the write buffer does not store parity bits. A 64-bit write is expanded as two 32-bit writes at the end of the outbound pipeline by the SSRAM state machine. Both writes may be issued for a 64-bit write before the state machine responds to a PM request. If the write buffer is full, it may take up to 16 clock cycles to dump the contents to an external SSRAM. It may take longer if there are PM accesses in between. PM 126 may interrupt a burst of writes at any time and may be delayed an additional cycle during the beginning of a 64-bit write.

Detail Description Paragraph:
[0225] Burst writes are handled in an opposite manner where only one acknowledge is returned on the GAB. A burst write enters the outbound pipeline, travels to the inbound pipeline, is written to the write buffer, and finally drops the associated tag that contains information so it does not enter the In FIFO. Once the write data for the end-of-burst write enters the write buffer, the information passes to the In FIFO. The burst write is acknowledged on the GAB to indicate the entire write

burst sequence has completed. A burst read or write to a register is acknowledged with a bus error.

Detail Description Paragraph:
[0227] Both pipelines may be accessed at the same time since PM 126 reads access both banks. As stated above, read data is obtained from external memory because the write buffer is flushed before the PM read occurs. PM reads always have priority over other operations in the pipelines and the SSRAM state machine stalls the outbound pipeline while PM reads are issued to external memory.

Detail Description Paragraph:
[0228] Atomic adds enter the outbound pipeline if there is room in the write buffer since they need to perform a write after it completes the addition. The MUXs issue the atomic adds into the Out FIFO unchanged and when they reach the output side of the Out FIFO, the address comparison logic treats it as a write by comparing the address with the addresses in the write buffer. If there is a match, tag bits are set to match the physical address in the write buffer. If there is a miss, the address is reserved similar to a write and the atomic add proceeds down the outbound pipeline.

Detail Description Paragraph:
[0229] The SSRAM state machine issues a read to memory and the data information enter the inbound pipeline. Just before the atomic add reaches the ALU, the MUX selects the most recent data, whether it is from memory (most likely), the write buffer, or from one of the pipeline stages ahead of the ALU. The feedback paths ahead of the ALU are necessary to handle back-to-back atomic operations to the same address without stalling the inbound pipeline.

Detail Description Paragraph:
[0230] The ALU performs the bit addition based on the bit field settings and modifies the result to all 1's if the carry bit asserts and the operation is sticky. Subtraction occurs when the supplied data is negative, in 2's compliment form. In this case, if the result "rolls over" from a negative number to a positive number and the operation is sticky, the result is also modified to all 1's. The result from the ALU is always stored in the write buffer and external memory as a 2's compliment number. This means that the most significant bit indicates the sign, leaving the remaining n-1 bits to indicate the value. When the atomic add is acknowledged, the tag information field is updated accordingly. Atomic adds to a register may not be issued to the pipeline since the MUX may simply return a bus error on the GAB slave return bus.

Detail Description Paragraph:
[0231] Statistic adds may be submitted to the outbound pipeline the same way as atomic adds. The only difference is how the ALU processes them. The memory location represents a 64-bit quantity and the value added is a 32-bit quantity. The MUX just before the ALU may use the most recent 64-bit result as with atomic adds. The 64-bit result may be positive and added with the 32-bit quantity which has 2's compliment form. Once the addition has completed, the 64-bit quantity may be written to the write buffer and sent to the In FIFO and finally to the GAB as a statistic add acknowledge cycle.

Detail Description Paragraph:
[0246] Flexible Data Output Buffer

CLAIMS:

1. A method for direct access to bit fields in instruction operands, the method comprising: providing a bit field consisting of a plurality of bits in a plurality of bit positions; performing instruction operations utilizing bit fields in source and target operands; and providing direct manipulation of any bits in any bit

field.

2. The method for direct access to bit fields in instruction operands according to claim 1, further comprising: transferring data from an input buffer to a packet task manager; dispatching the data from the packet task manager to an analysis machine; classifying the data in the analysis machine; and modifying and forwarding the data in a packet manipulator; wherein no instruction depends on a preceding instruction because each instruction in a pipeline is executed for a different thread.

3. The method for direct access to bit fields in instruction operands according to claim 1, further comprising: transferring the data after modifying and forwarding to an output buffer.

4. The method for direct access to bit fields in instruction operands according to claim 1, further comprising: processing data at a rate of at least 10 Gbs.

11. The apparatus according to claim 9, further comprising: packet input global access bus software code used for flow of data packet information from a flexible input data buffer to an analysis machine.

18. The apparatus according to claim 9, further comprising: a bidirectional access port operationally connected to said analysis machine; a flexible data input buffer operationally connected to said analysis machine; and a flexible data output buffer operationally connected to said analysis machine.

First Hit

Generate Collection Print

L15: Entry 1 of 3 File: PGPB Oct 2, 2003

DOCUMENT-IDENTIFIER: US 20030185243 A1
TITLE: Procedure and controller for the allocation of variable time slots for a data transmission in a packet-oriented data network

Abstract Paragraph:
The invention relates to a procedure for the allocation of time slots for a transmission of data in variable time slots between a controller of a packet-oriented data network on the one hand and a terminal of a user of the data network on the other hand. The time slots are assigned to a terminal for a data flow following request made by the terminal and in dependence on the transmission capacity available in the data network, and are temporarily stored in a ring buffer before they are allocated to the terminal at a predefinable allocation instant. In order to simplify and accelerate the allocation of time slots in such a data network, it is proposed that a temporal write range be defined in the ring buffer by a variable first upper time threshold and a variable first lower time threshold.

CLAIMS:

1. Procedure for the allocation of time slots for a transmission of data in variable time slots between a controller of a packet-oriented data network on the one hand and a terminal of a user of the data network on the other hand, the time slots for the data transmission being assigned to a terminal for a data flow following request made by the terminal and in dependence on the transmission capacity available in the data network, and being temporarily stored in a ring buffer before they are allocated to the terminal at a predefinable allocation instant, wherein in the ring buffer a temporal write range is defined by a variable first upper time threshold and a variable first lower time threshold, a position of a write pointer to a field of the ring buffer within the write range is determined in dependence on the allocation instant and the time slots assigned to the terminal for the data flow are stored in a field corresponding to the position of the write pointer.

6. Controller for a packet-oriented data network for the transmission of data in variable time slots between the controller on the one hand and a terminal of a user of the data network on the other hand, with means for receiving and processing a request made by a terminal for time slots, and with means for allocating time slots upon the request made by the terminal, the means for allocation assigning the time slots to a terminal in dependence on the transmission capacity available in the data network (1) and temporarily storing them in a ring buffer before the means for allocation allocate the time slots to the terminal at a predefinable allocation instant, wherein the means for allocation comprise: means for the definition of a temporal write range in the ring buffer by a variable first upper time threshold and a variable first lower time threshold; means for determining a position of a write pointer (37) to a field of the ring buffer within the write range in dependence on the allocation instant; and means for storing the time slots assigned to the terminal for the data flow in a field corresponding to the position of the write pointer.

First Hit Fwd Refs

Generate Collection

L6: Entry 11 of 18 File: USPT Dec 16, 2003

DOCUMENT-IDENTIFIER: US 6665755 B2
TITLE: External memory engine selectable pipeline architecture

Abstract Text (1):
External memory engine selectable pipeline architecture provides external memory to a multi-thread packet processor which processes data packets using a multi-threaded pipelined machine wherein no instruction depends on a preceding instruction because each instruction in the pipeline is executed for a different thread. The route switch packet architecture transfers a data packet from a flexible data input buffer to a packet task manager, dispatches the data packet from the packet task manager to a multi-threaded pipelined analysis machine, classifies the data packet in the analysis machine, modifies and forwards the data packet in a packet manipulator. The route switch packet architecture includes an analysis machine having multiple pipelines, wherein one pipeline is dedicated to directly manipulating individual data bits of a bit field, a packet task manager, a packet manipulator, a global access bus including a master request bus and a slave request bus separated from each other and pipelined, an external memory engine, and a hash engine.

Application Filing Date (1):
20001222

Brief Summary Text (2):
This invention generally relates to the field of data communications and data processing architectures. More particularly, the present invention relates to a novel external memory engine (EME) selectable pipeline architecture for a multi-thread packet processor which processes data packets using a multi-threaded pipelined machine wherein no instruction depends on a preceding instruction because each instruction in the pipeline is executed for a different thread.

Brief Summary Text (10):
Methods and apparatuses consistent with the principles of the present invention, as embodied and broadly described herein, provide an EME selectable pipeline architecture to a multi-thread packet processor that processes data packets using a multi-threaded pipelined machine wherein no instruction depends on a preceding instruction because each instruction in the pipeline is executed for a different thread. The multi-thread packet processor transfers a data packet from a flexible data input buffer to a packet task manager, dispatches the data packet from the packet task manager to a multi-threaded pipelined analysis machine, classifies the data packet in the analysis machine, modifies and forwards the data packet in a packet manipulator. The multi-thread packet processor includes an analysis machine having multiple pipelines, wherein one pipeline is dedicated to directly manipulating individual data bits of a bit field, a packet task manager, a packet manipulator, a global access bus including a master request bus and a slave request bus separated from each other and pipelined, an external memory engine, and a hash engine.

Detailed Description Text (4):
As shown in FIG. 1, an embodiment of the route switch packet architecture according to one aspect of the invention comprises Bi-directional Access Port (BAP) 10, Host

Packet Injection (HPI) 14, Flexible Data Input Buffer (FDIB) 20, Test 28, Clock & PLLS 30, Analysis Machines (AMs) 42, 56, 70, 84, Packet Task Manager (PTM) 98, Global Access Buses (GAB) 108, 110, 112, 114, 116, 118, External Memory Engines (EME) 120, 156, Internal Memory Engines (IME) 122, 152, Packet Manipulator (PM) 126, Hash Engine (HE) 158, Centralized Look-Up Engine Interface (CIF) 160, Flexible Data Output Buffer (FDOB) 162, and Search/Results/Private 166, 168. With the exception of Search/Results/Private 166, 168, the combination of the above described elements may be considered a multi-thread packet processor.

Detailed Description Text (17):
In a packet processor, there is no explicit relationship from one packet to another packet except for the sequence of packets. The packets may be dispatched to multiple processing units or to multiple threads on a pipelined processing engine, as long as the packet sequence is maintained. Because of this, the multi-thread packet processor may be partitioned into multiple packet processing units, each being multi-threaded to keep all execution pipelines fully operating. Since this is a hardware partitioning, the packet sequencing is kept in hardware via PTM 98. As previously mentioned, the multi-thread packet processor may be designed for up to 250 MHz with 4 packet processing units providing 16.5 MPPS with 60 instructions used per packet forwarding decision.

Detailed Description Text (18):
Because the multi-thread packet processor processes the packets, it includes search capabilities. A common search metric used is the number of lookups per second the processor is capable of performing. The metric is typically bound, so that relative performance can be measured. Lookups using the radix-4 method can be effectively used in the routing of IP packets. The number of 24-bit radix-4 lookups for the multi-thread packet processor is a direct relation of the number of memory accesses EMEs 120, 166 are able to do per second. (The lookup functionality is part of the External Memory Engine submodule.)

Detailed Description Text (21):
BAP 10 may be designed for access by a general-purpose processor. All memory and register locations in the multi-thread processor address space are accessible from BAP 10. In an effort to make BAP 10 adaptable to future requirements, BAP 10 may be available to AMs 42, 56, 70, 84 with the intention of reading status information from external peripheral devices. One application is the reading of external queue depths for use in implementing intelligent drop mechanisms. It is assumed that these algorithms only need to access the peripheral bus periodically. Thus, the interface can be shared with arbitrated host accesses. If host accesses are limited once a system is in a steady state, the multi-thread packet processor is capable of supporting accesses up to once per packet. At 16 million packets per second (MPPS), this equates to 16 million peripheral accesses per second. Thus, the multi-thread packet processor 250 MHz operation allows up to 15 cycles per access.

Detailed Description Text (22):
BAP 10 is configured as a shared multiplexed address and data bus that supports accesses to and from a generic host and peripheral devices. BAP 10 contains Global Registers 12, which include configuration and status registers that are global to the multi-thread packet processor. Registers that are specific to an element's function are contained in that element and accessible via one of the element's GAB interfaces. The operation of BAP 10 is controlled by BAP Global Registers 12. These registers include the source address, destination address, status register, interrupt vector, transfer size register, and several others. BAP's 10 interface to a host uses a chip select and ready control handshaking mechanism, allowing BAP 10 to interface with an external host operating at an unrelated asynchronous frequency. BAP 10 interfaces to all of the multi-thread packet processor's elements on each of the internal GABs 108, 110, 112, 114, 116, 118. BAP 10 provides direct accesses to all internal memory and register locations for normal read and write operation types.

Detailed Description Text (26):
Flexible Data Input Buffer

Detailed Description Text (28):
FDIB 20 also contains the main packet buffering for the multi-thread packet processor. FDIB 20 includes four Packet Memories 26. Each of these memories may be configured as a 512.times.128-bit dual port memory device that is segmented into 512 64-byte buffers. Each buffer has a page descriptor word contained in a separate 512.times.27 dual port memory. As pages fill, the descriptors are parsed and packet descriptors are generated with information including error-type (e.g., 3-bits), the length of the packet (e.g., 13-bits) as calculated by FDIB 20, and the master sequence number (e.g., 12-bits). Additionally stored are the receive port (e.g., 4-bits) and the address of the first page of the packet. All FDIB Packet Memories 26 and configuration registers are accessible by the host as well, with Packet Memories 26 being restricted to diagnostic mode access.

Detailed Description Text (39):
The state of each thread is independent from the state of all other threads. Threads and their register content are identified by a Thread Identification (TID) number. Status is provided to indicate which threads are active or inactive, enabled or disabled, etc. In addition to the AM integer pipeline that starts the execution of every AM instruction, each AM has access to several specialized co-processor units such as EMEs 120, 266, HE 158, etc. The TID follows the instruction everywhere in the AM or co-processor pipelines. The TID is also the primary mechanism of control between all co-processing units, packet data interfaces, packet pre-classifiers, and the integer pipeline. For most of the interfaces, a TID Queue is used. Each TID queue is 16.times.4 bit FIFO that contains the thread identifications for some particular operation. Some of the TID queues have multiple write ports to allow new, continued, or co-processor return operations to be started simultaneously.

Detailed Description Text (42):
Each of AMs 42, 56, 70, 84 have the following features: 1) Instruction based micro-coded processing. AMs 42, 56, 70, 84 do not hard code all operations for each packet protocol and therefore may operate on any changed or future protocol. 2) Robust instruction set with special networking applications based instructions such as lookup. All instructions are 3 argument. Assembler mnemonics provide 1/2 argument look and feel instructions. 3) Direct access of packet header memory. 4) Bit field operations. 5) Conditional execution. 6) Branching capabilities on all instructions. 7) Integer Pipeline. 8) Exception processing with external event generation. 9) Full packet memory access. 10) Results memory access. 11) Search Co-processor. 12) Statistics Co-processor. 13) High-speed Private Memory. 14) Hash Co-processor (HE 158). 15) CLUE I/F (CIF 160). 16) Peripheral access. 17) Hardware pre-classification.

Detailed Description Text (44):
Each AM includes packet pre-classification hardware. PTM 98 passes the length and address of the first buffer page of a packet to an AM thread. The next available thread takes the address and begins a fetch of the page into the Packet Header Memory contained in the AM. While the transfer is occurring over the AM's Packet Input GAB I/F, the pre-classification hardware snoops the data to classify the most basic known types. The hardware classification may be programmable and may be enabled or disabled. The concept of the hardware pre-classification is to aid the AM in a "fast dispatch" saving instructions for more critical processing. As such, pre-classification may be limited to well known protocols that make up 90-95% of the packet traffic. The pre-classification also aids in attempting to maintain line rate for packets smaller than 64-bytes. By pre-classifying some of the small packet types, less instructions can be used for these types, which in turn yields more processing power in the multi-thread packet processor and then the subsequent

support of line rate for these as well.

Detailed Description Text (50):
Each of the AM threads has a context of its own registers and so on. The registers and packet memory are physically in a shared memory between the threads, but their direct access and use by a single thread makes them private. The private resources are as follows: 1) Five address registers used to access packet and processing environment data. These registers are generally assumed to contain addresses. They are not normally used as temporary holding registers, as some implementations may assume that they hold valid addresses. 2) PTM Descriptor Memory Control--10 bit--points to 64-bit control structure in PTM for forwarding to PM, written by the PTM used by the AM hardware classification to fetch initial packet page into the Packet Header Memory and by the AM thread for the DONE issue. 3) Packet Input Pointer--13 bit--points to first 64-bit word of the inputted packet in the FDIB or PHI, written by the PTM used by the AM threads. The Packet Input Pointer should be copied to AO for access deeper in the packet. 4) Packet Header Memory Payload--6 bit--points to the first byte of payload data as determined by the hardware classification, written by the classifier and used by the AM threads. 5) Address Register 0 (A0)--13 bit--64-bit aligned address into packet memory located in FDIB or PHI, written and used by AM threads for deeper packet access. 6) Address Register 1 (A1)--6 bit--byte address into packet header memory for the thread, written and used by AM threads for packet analysis. 7) 8 64-bit general-purpose data registers that can be used as temporary variable storage or as address pointers for load or store instructions. 8) 8 64-bit result registers that are used for return data from the co-processing units. These may be used as source operands in all integer pipe operations but not as a destination. 9) D30/D31 addressing. 10) 11-bit program counter (PC), implicitly incremented or explicitly changed via flow control. 11) 7-bit condition code register with implicit/explicit setting by SETBRCC field of instruction and the result of the instruction. Condition code is used on subsequent instructions for conditional execution. 12) 6-bit FCFO Index Register. The FCFO instruction sets this register. 13) 64-bit Filter Accumulator Register. The FILTER instruction sets this register. 14) Additional special registers such as the 16-bit Thread Status Register. 15) 11-bit implicit link register set on SETBRCC instruction branch. Explicitly linking to the data registers or the link register is used for certain instructions. 16) 64-byte packet header memory.

Detailed Description Text (53):
The direct memory access of AMs 42, 56, 70, 84 however, may be limited to the Packet Header Memory (PHM) contained in each AM. The PHM is pre-loaded by an AM prior to starting a thread for packet processing, which is, coincidentally, when the AM performs the fast dispatch. The AM thread then has full access to the portions of the packet residing in the 64-byte PHM buffer. The 64-byte restriction is deemed sufficient as this will fit most known protocols with a reasonable descriptor attached. The AM thread also has the ability to go deeper in a packet with memory accesses from FDIB 20 or PHI to the PHM.

Detailed Description Text (55):
To combat this, the multi-thread packet processor allows direct manipulation of bit fields. The problem of bit field isolation, manipulation, and reintegration into the larger data item is handled by the underlying hardware rather than a sequence of instructions as would be done on a general purpose processor. The additional hardware increases the processing pipeline depth of each AM, but does not have a detrimental effect on the multi-thread packet processor throughput. For example, consider the problem of incrementing a 5-bit field within a word. The general-purpose processor generally needs to extract the field into a register, increment that register, and insert the field back. For AMs 42, 56, 70, 84, this function is effected using a single instruction: addD1 [field], 1,D0[field]

Detailed Description Text (75):
As shown in FIG. 3, the GAB is configured as a fully synchronous split operation

protocol that is separated into two sections: Master Request Bus (MRB) 306, 310 and Slave Result Bus (SRB) 308, 312. Each operation starts with a master request and an MRB arbiter 302 grant. The MRB registers the operation to the slave devices. The operation is completed by a slave request and SRB arbiter 304 grant. The SRB registers the data back to the masters. The MRB and SRB are separated from each other and are pipelined. This allows multiple master requests to fill the pipelines of the slave devices, which are typically co-processing units, and then wait for the return data. Since the multi-thread packet processor master devices are typically multi-threaded, multiple pipelined requests may occur from any given master. Each slave and master has a ready signal to indicate that it is ready for the next operation. Masters assert their ready to the SRB arbiter and slaves assert their ready to the MRB arbiter. It is up to the designer of the master or slave device to insure that the ready signal is only asserted when the device is ready for the operations of which it is capable. For example, if a GAB device typically takes burst writes, then the ready signal should be asserted when there is enough room for a burst. Since the arbiter knows which device a master wants to target and has the slaves ready, an additional level of arbitration can implicitly be built in by not granting a master the GAB if the targeted slave is not ready. Similarly, the SRB can implicitly hold off a slave for return data if the master to return data to is not ready. This should not occur since the master had originally requested the operation.

Detailed Description Text (76):
The GAB Arbiter MUX (GAM) 300 submodule contains all the logic necessary for both the MRB and SRB: the arbiters, address/data/control MUXes, registers, and buffers.

Detailed Description Text (78):
Each GAM is composed of six submodules: 1. MRB Arbiter 2. MRB MUX 3. MRB Register Buffer 4. SRB Arbiter 5. SRB MUX 6. SRB Register Buffer

Detailed Description Text (79):
The arbiters take the respective requests, readies, and the arbitration algorithm and grant a master (MRB) or slave (SRB) access to the split portion of the bus. The MRB MUX accepts select control from the MRB arbiter and multiplexes the various master signals to the MRB Register Buffer. All signals to the MRB from the masters should be registered outputs. The only incurred delay is the multiplexer structure. A single flip-flop for each data/address/control bit is provided in the MRB Register Buffer. Individual outputs with buffers are provided for each slave that needs a connection. The SRB Arbiter, SRB MUX and SRB Register Buffer work exactly the same as the MRB, except the operation types may be slightly different and the transfer is from one of the slave devices to one of the master devices.

Detailed Description Text (100):
Most operations on the MRB are single cycle since only a request needs to be transferred. The MRB registers and buffers the data to the slave being accessed. The MRB asserts the write signal to the slave, strobing in the request data. The slave performs the operation internal to its bounds. It then drives the appropriate return data, operation type, operation qualifier, master device/sub-device to return data to and address on its SRB GAB signals and asserts a request. The SRB arbiter eventually grants access to the slave. The grant is based on the other slave requests, the master ready, and the arbitration algorithm implemented. The SRB registers and buffers the operation return data to the master over the appropriate amount of cycles i.e., a burst read of 4 has 4 return data cycles at the master. Multiple slave destinations are allowed. Furthermore, the interleaving of slave return data on the SRB from two unique slaves back to one or more masters is also allowed and operates exactly the same. The MRB arbiter performs the operation and begins granting cycles based on the arbitration scheme and whether the requested slave is ready or not. A fair arbitration scheme is assumed, as well as the slave being ready. The 1st cycle is granted to master 0, the 2nd to master 1, the 3rd to master 0, and the final cycle to master 1. Since no one is

requesting, master 1 also gets the inadvertent grant which gets suppressed by the master asserting NOP, i.e. no write to the slave. The slave begins appropriate return operations to the master that requested it, by asserting its signals and requests. The latency of the response is dependent on the slave and the operation.

Detailed Description Text (103):
Packet Input GAB 106 provides an interface between AMs 42, 56, 70, 84 and FDIB 20. An AM pulls the first buffer of the packet from FDIB 20 into the Packet Header Memory of the AM. During the initial transfer, as noted above, the AM Hardware Pre-Classifier snoops the packet and provides information to the AM thread. Subsequent accesses deeper into the packet are under full thread control through a predetermined instruction. Packet Input GAB 106 is one of the GABs in the multi-thread packet processor used for the flow of packet data. Packet Input GAB 106 transfers packet data from FDIB 20 to one of AMs 42, 56, 70, 84. Typically, the transfer is the first page of a packet, but AMs 42, 56, 70, 84 may access any number of words to the maximum burst in order to look deeper into a particular packet, if the protocol dictates. Packet Input GAB 106 has as its bus masters: all four AMs 42,56,70, 84 and its slaves: FDIB 20 and HPI 14 submodules. The Packet Input GAB MRB uses TDMr arbitration. This allows fair access among AMs 42, 56, 70, 84 while not starving BAP 10. Each AM is allocated one out of every four cycles. BAP 10 is given 4 out of 256 possible time slices of the TDM and is the default member of the round robin i.e., BAP 10 wins round robin only if no AM is requesting.

Detailed Description Text (105):
Control GAB 108 provides an interface between an AM and PTM 98. PTM 98 transfers packet length, input port, and the address of the first packet buffer in FDIB 20 of the packet. The AM is configured as both a master and a slave on Control GAB 108. The registers/memories of the AM are accessible via Control GAB 108 by BAP 10. The multi-thread packet processor uses Control GAB 108 or the flow of control information between various masters of the multi-thread packet processor. It is used primarily for packet notification, sequencing, and internal descriptor (message) passing. Control GAB 108 is also used for programming the instruction memories and configuration information into the AMs 42, 56, 70, 84, PM 126 and PTM 98. Control GAB's 108 bus masters are: all AMs 42, 56, 70, 84, PTM 98 and BAP 10 submodules. Control GAB's 108 slaves are: all AMs 42, 56, 70, 84, PTM 98, PM 126, FDIB 20 and HPI submodules.

Detailed Description Text (121):
Each EME may access two separate 36-bit memory banks independently. Even parity is implemented for each byte during 32 and 64-bit accesses. An AM 40, 56, 70, 84 may issue burst reads or writes of 8 over the GAB, while PM 126 may issue a burst read of 8 using a separate high-speed access port with a 64-bit wide data bus. A 64-bit access from PM 126 is split across both banks, while a similar access from an AM 40, 56, 70, 84 may cause the EME to double pump a single bank. 64-bit accesses from an AM and PM 126 may be word aligned. Each bank supports memory expansion for 4 SSRAMs, 256K in depth each. When memory is expanded, a register bit is set to remove a pipeline stage in the read and write paths. This means the issue and receive queues are "shortened" by one entry so there is an extra clock cycle available to perform external address decode and data MUXing. The data bus MUXing between memories may be accomplished with transceivers so that all connections are point-to-point to run the memories at 200 MHz. When the EME is used in the CLUE, the signals for the high-speed access port may be tied off and the related logic optimized out during synthesis.

Detailed Description Text (122):
As such, lookup performance may reach 25+million lookups per second for a single memory bank. This is based on a 24-bit key for a IP-V4 lookup, requiring 7-8 memory cycles running at 200 MHz. Assuming the lookup tables are well distributed across memory banks, an EME could reach 50+million lookups per second using both memory

banks. Lookup performance is reduced when these operations have to compete with reads/writes from AMs 40, 56, 70, 84 and PM 126. Each EME is pipelined to improve speed and mirror external memory. There are two pipelines, one for each bank that operate independently of each other. Hence, there are two separate arithmetic and logic units (ALUs), two write buffers, etc.

Detailed Description Text (125):
The high-speed access port (HSAP) controller contains asynchronous FIFOs and control logic to handle burst reads from PM 126. The HSAP controller increments the address for burst reads so they appear as single reads to the EME pipelines. Both pipelines operate on the read request simultaneously since a PM read is normally 64-bits wide and split across both banks. The two data streams are merged at the PM outbound FIFO. Parity is checked, if enabled. If a parity error is detected, a parity error signal is asserted to PM 126 and to BAP 10. The write buffer in each pipeline may not used for PM accesses because the software may force a write buffer flush before PM 126 accesses the data. During the final write from an AM, a field may specify a flush operation. When the buffer is flushed, this write may be acknowledged on the GAB slave return bus so the AM knows the data is in external memory. The AM may then launch a job packet to PM 126, which can then access the data from external memory. The control logic in the HSAP controller handshakes with the SSRAM state machines in each pipeline so that all PM requests can be serviced immediately. The HSAP controller also contains a request FIFO to absorb multiple read requests to remove any bandwidth penalty associated with handshaking across an asynchronous boundary.

Detailed Description Text (126):
The input and output FIFOs buffer data flow between the pipelines and the MUXs. Since lookups and filters can be forwarded from one bank to the other (depending on the contents of the bank forwarding registers), a lockout condition can occur where the output FIFOs for each bank are full and each input FIFO has a lookup that needs to continue in the "other" bank. This is controlled by the MUXs that do not allow more than 32 operations to be submitted across both pipelines. The input FIFOs are 32 deep so, regardless how the operations flow through the pipelines and FIFOs, all operations can be absorbed by either input FIFO during a stall condition (PM access) so all lockout scenarios are avoided. A 64-bit access is counted as two operations and burst accesses are handled similarly. The MUXs increment a counter whenever something is entered into either output FIFO and is decremented whenever something is taken from either input FIFO that is destined for the GAB. The EME pipeline is a complex configuration that contains a Write Buffer, an ALU, and a Loopback FIFO. The EME directly controls external SSRAM, and services PM requests. The write buffer consists of a 64-bit wide by 8-deep memory to store data along with a "parallel" set of flops that store a 20-bit address, a pair of valid (V) bits, and a pair of reserved (R) bits. The write buffer can behave as a cache since the address of all requests from the output FIFO are compared with the write buffer addresses. However, this is not the main purpose of the buffer because most addresses to memory have random behavior, minimizing the probability of a hit in the write buffer. The main goal of the write buffer is to reduce bus turnaround time penalties by writing the data as a burst during a flush sequence. Since the buffer may contain eight 64-bit entries, this could take up to 16 clock cycles. If there is a PM access during a flush, the SSRAM state machine stalls the write buffer flush, turns the bus around to read data for PM 126, then turns the bus around again to finish the write buffer flush.

Detailed Description Text (127):
A write buffer flush is triggered under the following circumstances: (1) a write is present in the Output FIFO and the write buffer is full; (2) the write buffer flush register bit is set; (3) a write is issued with bit 2 of a field set. When a write buffer flush is in progress, the write that is present in the Out FIFO is also sent to memory before the bus is turned around for reads. The write with flush option may be used when updating lookup tables and PM data structures. This is because

addresses are compared at the Out FIFO and not at the Loopback FIFO or at the PM interface. Neglecting to flush the write buffer may cause PM 126 to retrieve "stale" data from external memory.

Detailed Description Text (128):
The ALU performs all the arithmetic functions for atomic and statistical adds, including the "stickiness" feature, as well as address calculation for lookups and filters. When there is a hit in the write buffer while an atomic or statistical add is issued from the Out FIFO, a memory cycle is wasted as the operation travels from the outbound pipeline to the inbound pipeline, dropping the read data from external memory to use the data in the buffer instead. A large MUX before the ALU controls data flow, selecting the most recent data during back-to-back atomic operations using the same address. Output and input delay cells are added to improve setup/hold times in the read/write paths to external memory. There is a 2-to-1 MUX to select data for memory writes, using a memory control signal from a register bit. This signal is low when late-write SSRAMs are used, so the data is driven one clock cycle after the address. If a different memory is used where data must be driven two cycles after (i.e., burst mode SSRAMs), a register bit can be set to flip the MUX to select data from the next stage in the pipeline. Similarly, a MUX using a memory expansion signal selects which address and associated tag information corresponds to the incoming SSRAM data. When expanded memory is used, the address is delayed a couple clock cycles to match the extra external delay where one additional clock cycle is allowed for external address decode and data MUXing, and a second extra cycle to register the read data externally before it is supplied to the multi-thread packet processor.

Detailed Description Text (129):
The pipeline flow is best explained by describing how basic operations travel through the various stages. Before discussing how reads are processed, it is important to understand how the SSRAM state machine controls the flow of data from the Out FIFO, the Loopback FIFO, the Write Buffer during flushes, and all PM requests. The SSRAM state machine can stall the outbound pipeline that is fed from the Out FIFO whenever there is a PM access. Conversely, the inbound pipeline that contains the ALU cannot be stalled. Whenever the outbound pipeline is stalled, all operations from the inbound pipeline are absorbed by the Loopback FIFO or Input FIFO. If a read is supplied by the Out FIFO, the SSRAM state machine allows it to enter the outbound pipeline if there are no PM accesses, the Loopback FIFO is empty, and no write buffer flush is in progress. If there is an address match with one of the entries in the write buffer, a tag bit is set along with a 4-bit address that corresponds to the physical address in the write buffer. The upper three address bits are used to supply the address while the lowest bit selects the upper or lower 32-bit word when the read reaches the write buffer. For a 36 or 64-bit read, this lowest address bit is ignored. Once the read enters the outbound pipeline, the address is driven on the next cycle while the same address and related tag information travels in the pipeline.

Detailed Description Text (130):
Once the address/tag reaches the inbound pipe, the read data from the SSRAM arrives at the same cycle so they travel as a pair up the inbound pipe, destined for the PM, In FIFO, or Loopback FIFO. If the read is for a 64-bit access, the SSRAM state machine may issue the original word-aligned address during the first cycle, and then reissue the same address with bit 0 set to access the upper 32 bits of the 64-bit word. The state machine stalls the outbound pipeline during this second access. When the data arrives at the inbound pipeline, each 32-bit word is parity checked if parity checking is enabled. The two 32-bit words then enter a MUX to be merged as a single 64-bit word (single stage) before the data is issued to the ALU. If there were a hit in the write buffer, this data would have been MUXed instead, dropping the data from external memory. For a read, the ALU simply passes the data along to the In FIFO, destined for the GAB. A write operation travels down the outbound pipeline if there is room in the write buffer and the buffer is enabled.

As always, the SSRAM state machine gives priority to entries in the Loopback FIFO so it must be empty before any operations are accepted from the Out FIFO. The write address may be stored in the write buffer and marked as reserved. If its a 64-bit write, two reserve bits may be set to indicate that the upper and lower 32 bits will be stored. The write may be issued, but tagged as a read in the pipeline so the SSRAM state machine does not turn the bus around. Once the address/tag reaches the inbound pipeline, the data is dropped because the tag information is still a write.

Detailed Description Text (131):
When the address/tag/write data reaches the ALU, the data is written to the write buffer and the entry is marked as valid (2 valid bits for a 64-bit write). At the same time, the write is returned to the In FIFO so a write acknowledge can be sent out the GAB. If parity is enabled and a write buffer flush occurs, all 32 and 64-bit write data is passed through the parity generation logic. A 64-bit write is stalled in the pipeline until the upper and lower 32-bit words are written to memory, using the MUX to steer the data to the final output flops. If a 64-bit write were destined for the write buffer instead, the parity generation logic is bypassed and a single read cycle is wasted as the data is dropped at in inbound pipeline. During a write buffer flush, none of the writes return to the GAB as write acknowledges since they were already sent when the write buffer was initially loaded.

Detailed Description Text (132):
Atomic and statistical adds operate in a similar manner to writes, reserving an address in the write buffer. If the buffer is full, it is flushed before the add is taken from the Out FIFO. The read data from external memory is parity check (32 or 64 bit access) and supplied to the MUX in the inbound pipeline. If the address in the write buffer is only reserved and not valid, the read data from memory is supplied to the ALU for the addition. In the case of back-to-back atomic operations to the same address, the most recent data is ahead of the ALU and not necessarily in the write buffer. In this case, the MUX selects the data after the ALU in one of the pipeline stages, instead of the data from external memory. Once an atomic operation passes through the ALU, the data is written to the write buffer and the result is also passed to the In FIFO to issue an acknowledge cycle on the GAB.

Detailed Description Text (133):
Lookups and filters enter the outbound pipeline the same way as reads, however, the address is not compared with the addresses in the write buffer. Lookups and filters are treated as reads in the outbound pipeline and the read data arrives at the ALU in the inbound pipeline. The ALU calculates the lookup address if the continue bit is set and the new address and remainder of the key travel into the Loopback FIFO if the lookup needs to proceed in the same bank and same EME. The bank forwarding registers control the flow of lookups and filters. Later sections in this chapter provide a more thorough explanation of lookups and filters.

Detailed Description Text (135):
In one implementation, the difference between MEMCLK and the return clock should not exceed 2nS since there is no logic between the first and second stage of flops. During 32 and 64-bit read operations, parity is verified if enabled by setting a bit in the configuration register. If a parity error occurs, a bit is set in the status register and a parity error signal is asserted to BAP 10. A 64-bit read double pumps a single memory and is always word aligned, meaning address bit zero is ignored. If the write buffer is enabled, the read address is compared with the addresses stored in the write buffer. Regardless of a hit or miss, the read travels down the outbound pipeline to initiate a SSRAM read. The read data is replaced with the data in the write buffer or from one of the feedback paths after the ALU. This decision is done by the MUX block one pipeline stage before the ALU. The read proceeds to the In FIFO, and then finally to the GAB using the device and sub-device information that travels with the read through the pipeline.

Detailed Description Text (136):
A read with clear tag information is treated as an atomic operation, except the original value is sent to the GAB while a value of all zeros is written to the write buffer. A register read never enters the pipeline since all registers are present in the MUX. Register reads are responded to immediately and do not follow the order of operations in the pipeline. They remain ordered compared to other register accesses and are never inhibited unless the GAB slave return bus is stalled. If the pipelines are disabled via a register bit, any operations destined for either pipeline are dropped so reads can pass through the master request GAB FIFO and complete without interruption.

Detailed Description Text (137):
Write operations have the same address mapping as reads, and address matching is applied if the write buffer is enabled. The write travels down the outbound pipeline as a read, and the read data is dropped in the inbound pipeline. The write data is written to the reserved location in the write buffer after passing through the ALU. Parity is not generated until the write buffer is flushed since the write buffer does not store parity bits. A 64-bit write is expanded as two 32-bit writes at the end of the outbound pipeline by the SSRAM state machine. Both writes may be issued for a 64-bit write before the state machine responds to a PM request. If the write buffer is full, it may take up to 16 clock cycles to dump the contents to an external SSRAM. It may take longer if there are PM accesses in between. PM 126 may interrupt a burst of writes at any time and may be delayed an additional cycle during the beginning of a 64-bit write.

Detailed Description Text (139):
Burst writes are handled in an opposite manner where only one acknowledge is returned on the GAB. A burst write enters the outbound pipeline, travels to the inbound pipeline, is written to the write buffer, and finally drops the associated tag that contains information so it does not enter the In FIFO. Once the write data for the end-of-burst write enters the write buffer, the information passes to the In FIFO. The burst write is acknowledged on the GAB to indicate the entire write burst sequence has completed. A burst read or write to a register is acknowledged with a bus error.

Detailed Description Text (141):
Both pipelines may be accessed at the same time since PM 126 reads access both banks. As stated above, read data is obtained from external memory because the write buffer is flushed before the PM read occurs. PM reads always have priority over other operations in the pipelines and the SSRAM state machine stalls the outbound pipeline while PM reads are issued to external memory.

Detailed Description Text (142):
Atomic adds enter the outbound pipeline if there is room in the write buffer since they need to perform a write after it completes the addition. The MUxs issue the atomic adds into the Out FIFO unchanged and when they reach the output side of the Out FIFO, the address comparison logic treats it as a write by comparing the address with the addresses in the write buffer. If there is a match, tag bits are set to match the physical address in the write buffer. If there is a miss, the address is reserved similar to a write and the atomic add proceeds down the outbound pipeline.

Detailed Description Text (143):
The SSRAM state machine issues a read to memory and the data information enter the inbound pipeline. Just before the atomic add reaches the ALU, the MUX selects the most recent data, whether it is from memory (most likely), the write buffer, or from one of the pipeline stages ahead of the ALU. The feedback paths ahead of the ALU are necessary to handle back-to-back atomic operations to the same address without stalling the inbound pipeline.

Detailed Description Text (144):
The ALU performs the bit addition based on the bit field settings and modifies the result to all 1's if the carry bit asserts and the operation is sticky. Subtraction occurs when the supplied data is negative, in 2's compliment form. In this case, if the result "rolls over" from a negative number to a positive number and the operation is sticky, the result is also modified to all 1's. The result from the ALU is always stored in the write buffer and external memory as a 2's compliment number. This means that the most significant bit indicates the sign, leaving the remaining n-1 bits to indicate the value. When the atomic add is acknowledged, the tag information field is updated accordingly. Atomic adds to a register may not be issued to the pipeline since the MUX may simply return a bus error on the GAB slave return bus.

Detailed Description Text (145):
Statistic adds may be submitted to the outbound pipeline the same way as atomic adds. The only difference is how the ALU processes them. The memory location represents a 64-bit quantity and the value added is a 32-bit quantity. The MUX just before the ALU may use the most recent 64-bit result as with atomic adds. The 64-bit result may be positive and added with the 32-bit quantity which has 2's compliment form. Once the addition has completed, the 64-bit quantity may be written to the write buffer and sent to the In FIFO and finally to the GAB as a statistic add acknowledge cycle.

Detailed Description Text (160):
Flexible Data Output Buffer

CLAIMS:

1. A method for providing external memory services to a multi-thread packet processor comprising: decoding information data sequences for a data packet within the multi-thread packet processor to determine assigning an operation to one of a first external memory bank and a second external memory bank; executing a read operation by comparing a read address with addresses stored in a write buffer, initiating a read, and replacing read data with data in the write buffer; executing a write operation by comparing a write address with addresses stored in a write buffer when the write buffer is enabled, and writing data to a reserved location in the write buffer; executing an atomic add operation by performing a write operation in the write buffer if there is available space in the write buffer; executing a lookup operation by indexing into a memory bank based on a base address, a key, and a key length, and conducting flow and memory accesses of the memory bank; and returning a result data field to the multi-thread packet processor when one operation is completed.

3. An apparatus for providing external memory services to a multi-thread packet processor, said apparatus comprising; a packet manipulator to decode information data sequences for a data packet within the multi-thread packet processor to determine assigning an operation to one of a first external memory bank and a second external memory bank; a write buffer to execute a read operation by comparing a read address with addresses stored in the write buffer, initiating a read, and replacing read data with data in the write buffer; and to execute a write operation by comparing a write address with addresses stored in the write buffer when the write buffer is enabled, and writing data to a reserved location in the write buffer; an arithmetic and logic unit operationally connected to said write buffer to perform an atomic add operation by performing a write operation in the write buffer if there is available space in the write buffer; and a loopback first-in-first-out unit operationally connected to said write buffer to return a result data field to the multi-thread packet processor when one operation is completed.

4. An apparatus for providing external memory services to a multi-thread packet

processor according to claim 3, further comprising: an analysis machine having multiple pipelines, wherein one pipeline is dedicated to directly manipulating individual data bits of a bit field; and a packet task manager operationally connected to said analysis machine.

8. The apparatus according to claim 7, further comprising: packet input global access bus software code used for flow of data packet information from a flexible input data buffer to an analysis machine.

15. The apparatus according to claim 7, further comprising: a bi-directional access port operationally connected to said analysis machine; a flexible data input buffer operationally connected to said analysis machine; and a flexible data output buffer operationally connected to said analysis machine.

First Hit
Search Forms
Search Results
Generate Collection
Help
User Searches
Preferences
Logout

Document 1 of 1 File: PGPB May 9, 2002

DOCUMENT-IDENTIFIER: US 20020054594 A1
TITLE: Non-blocking, multi-context pipelined processor

Application Filing Date:
20010830

Summary of Invention Paragraph:
[0002] This invention relates generally to packet switching controllers in a data network, and more particularly, to maximizing usage of a pipelined packet processor in processing incoming data packets.

Detail Description Paragraph:
[0022] The packet processor 12 preferably includes multiple sub-processors pipelined in series and operating independently of each other. The independent operation of the sub-processors allows the concurrent processing of different packet data at different stages of the pipeline. In addition, each sub-processor is also internally pipelined to allow current processing of multiple instructions associated with one or more packets within each stage.

L15: Entry 2 of 3 File: USPT Nov 19, 2002

DOCUMENT-IDENTIFIER: US 6483820 B1
TITLE: System and method for dynamic radio resource allocation for non-transparent high-speed circuit-switched data services

Detailed Description Text (12):
Associated with buffers 350A and 350B are, in a preferred embodiment, a plurality of network operator definable thresholds 390 and status flags 391, as illustrated in more detail in FIG. 4. These thresholds are, respectively, a buffer capacity threshold 392 and a buffer capacity rate change threshold 394. The status flags include a resource restriction flag 396, a traffic channel availability flag 398 and a maximum transfer rate flag 400, respectively. Buffer capacity threshold 392 can, for example, be set to declare an upgrade capacity XX associated with buffers 350A and 350B, e.g., a percentage available of the maximum capacity or a numerical representation of the bytes available in one or both of the buffers, that when reached will prompt the GIWU 370 to request the MSC/VLR 340 to allocate an additional channel for the MS 310.

WEST Search History

Hide Items | Restore | Clear | Cancel

DATE: Sunday, March 21, 2004

Hide?	Set Name	Query	Hit Count
	DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=ADJ		
☐	L6	20020225	30
☐	L5	L3 and ((pipeline or pipelining or pipelined) near8 packet near8 processing)	38
☐	L4	L3 and l2	0
☐	L3	buffer near8 (manager or management)	9763
☐	L2	20020225	17
☐	L1	pipeline near8 (synchronize or synchronization) near8 signal near8 processor	19

END OF SEARCH HISTORY

L24: Entry 5 of 21 File: USPT Jan 14, 2003

DOCUMENT-IDENTIFIER: US 6507898 B1
** See image for Certificate of Correction **
TITLE: Reconfigurable data cache controller

Application Filing Date (1):
19980218

Detailed Description Text (23):
Modern computer systems typically require some method of memory management to provide for dynamic memory allocation. In the case of a system with one or more co-processors, some method is necessary to synchronize between the dynamic allocation of memory and the use of that memory by a co-processor.

Detailed Description Text (550):
1) The processing unit requests a packet of data by supplying an address to the processing unit interface of the cache controller 1878; 2) Each of the eight address generator units 1881 then generate a separate address for each block of cache memory depending on the mode of operation; 3) The Tag portion of each of the generated addresses is then compared to the Tag address stored in the four blocks of triple-port Tag memory 1886 and addressed by each of the corresponding line part of the eight generated addresses; 4) If they match, and the line valid status 1873 for that line is also asserted, the data requested for that block of memory is deemed to be resident in the said cache memory 230; 5) Data that is not resident is fetched via the external bus 1890 and all eight blocks of the cache memory 230 are updated with that line of data from external memory. The Tag address of the new data is then written to the Tag memory 1886 at the said line address, and the line valid status 1873 for that line asserted; 6) When all requested data items are resident in cache memory 230, it is presented to the processing unit in a predetermined packet format.

First Hit Fwd Refs

Generate Collection

L24: Entry 15 of 21 File: USPT May 22, 2001

DOCUMENT-IDENTIFIER: US 6237079 B1
** See image for Certificate of Correction **
TITLE: Coprocessor interface having pending instructions queue and clean-up queue and dynamically allocating memory

Application Filing Date (1):
19980218

Brief Summary Text (4):
Modern computer systems typically require some method of memory management to provide for dynamic memory allocation. In the case of a system with one or more co-processors, some method is necessary to synchronize between the dynamic allocation of memory and the use of that memory by a co-processor.

Brief Summary Text (10):
In accordance with one aspect of the present invention, there is disclosed a method of controlling the interaction between a host CPU and at least one co-processor in a computer system to permit substantially simultaneous decoupled execution of CPU instructions and co-processor instructions, and dynamic allocation of commonly used memory space during the course of the execution of said instructions, said method comprising the steps of:

Brief Summary Text (17):
(b) a memory manager means connected to said memory and said instruction generator means to dynamically allocate space in said memory for co-processor use in executing said sequence of co-processor instructions,

Detailed Description Text (63):
Modern computer systems typically require some method of memory management to provide for dynamic memory allocation. In the case of a system with one or more co-processors, some method is necessary to synchronize between the dynamic allocation of memory and the use of that memory by a co-processor.

Detailed Description Text (804):
1) The processing unit requests a packet of data by supplying an address to the processing unit interface of the cache controller 1878;

CLAIMS:

1. A method of controlling the interaction between a host CPU and at least one co-processor in a computer system to permit substantially simultaneous decoupled execution of CPU instructions and co-processor instructions, and dynamic allocation of commonly used memory space during the course of the execution of said instructions, said method comprising the steps of:

(a) said host CPU allocating memory resources to be utilized by a set of instructions to be co-processor executed;

(b) generating a queue of pending co-processor instructions to be executed and a clean up queue of co-processor instructions for which execution has been completed;

(c) from time to time, under control of said host CPU, releasing for reallocation memory resources previously utilized by the instructions contained in said clean up queue of executed instructions.

10. Dynamic memory management means in a computer system having a memory of predetermined size, a host CPU and at least one co-processor, said memory management means comprising:

(a) an instruction generator means connected with said host CPU and generating a sequence of instructions intended for co-processor execution,

(b) a memory manager means connected to said memory and said instruction generator means to dynamically allocate space in said memory for co-processor use in executing said sequence of co-processor instructions,

(c) a queue manager means connected to said instruction generator means, said memory manager means and said co-processor, said queue manager means being arranged to store said sequence of instructions in a queue of pending instructions to be co-processor executed and a clean up queue of instructions which have been co-processor executed,

wherein from time to time said queue manager means removes executed instructions from said clean up queue to thereby release for reallocation memory space previously allocated to said removed executed instructions.

First Hit Fwd Refs

Generate Collection

L24: Entry 19 of 21 File: USPT Feb 11, 1997

DOCUMENT-IDENTIFIER: US 5602995 A
TITLE: Method and apparatus for buffering data within stations of a communication network with mapping of packet numbers to buffer's physical addresses

Application Filing Date (1):
19940513

Brief Summary Text (13):
Elements are added to the transmit queue by the software driver whenever it needs to transmit information. Elements are removed from the transmit queue after successful transmission is assumed. Removal of the elements can be done either by the low-level software driver or by the communication controller. Elements are added to the receive queue by the communication controller whenever a relevant packet is received, and are removed by the low-level software driver upon processing the packet.

Brief Summary Text (25):
It is a further object of the present invention to provide such a method and apparatus of buffering data packets in a communication controller, in which data packet buffer memory is dynamically allocated so as to optimize memory utilization without burden on the host processor.

Brief Summary Text (28):
A further object of the present invention is to provide such a communication controller, in which dynamic allocation of buffer memory is transparent to the host processor and the medium access control unit of the communication controller.

Brief Summary Text (31):
According to one of the broader aspects of the present invention, a method of buffering data packets in a communication controller is provided. In the illustrated embodiments, the communication controller is interfaced with a processor for processing data packets, and includes a control unit for accessing a communication medium.

L2: Entry 13 of 17 File: USPT Dec 6, 1994

DOCUMENT-IDENTIFIER: US 5371896 A
TITLE: Multi-processor having control over synchronization of processors in mind mode and method of operation

Application Filing Date (1):
19930517

Drawing Description Text (42):
FIG. 41 is a graph of waveforms of the pipeline sequence for a synchronized processor waiting for a synchronization signal;

DOCUMENT-IDENTIFIER: US 20020110122 A1
TITLE: Dynamic packet processor architecture

Application Filing Date:
20010309

Detail Description Paragraph:
[0024] Depending on the input port type and the packet type, the packet processor 24 needs to perform specific filtering tasks in a given order at each processing stage of the packet Therefore, every packet flow has a predefined processing path with different pipeline stages and lookup numbers. To be able to handle all types of packet and input ports listed in the design requirements, without having to build complicated multi-branch pipelines, the generic pipeline stage assembly 26 can perform all of the required filtering tasks.

Detail Description Paragraph:
[0048] FIG. 5 is a schematic block diagram of the packet memory 32 including a queue management unit 148 and a memory management unit 150. The queue management unit 148 handles buffer allocation and management for packet payload and deals with a descriptor's head pointer list 161 and a descriptor's tail pointer list 163, as explained below, to determine what is the beginning and the end of every available queue. The descriptors are fixed in size and provide a link to the payload buffer. The descriptors are requested by the last pipeline stage 26n when a task is ready to be written into the packet memory 32.

Detail Description Paragraph:
[0051] A packet with a descriptor may be received by the queue management unit 148 from the last generic pipeline stage in the pipeline assembly 26 via the RFQI bus 40. The task is written and stored inside a buffer of the queue management unit 148 and a suitable descriptor will be developed that is stored next to the task. The descriptor will have information about at which address the task is stored and its length so that the output line interface can easily find the task via the descriptor in the unit 148. Additionally, each descriptor contains a pointer to the next descriptor's location. In this way, every active queue will have a chain of descriptors with first and last descriptors' pointers stored in head and tail pointer list, respectively. Each output channel/queue has its own descriptor chain. The output line interface may have multiple queue accesses. Each queue has a chain of descriptors, so when one descriptor is read and completed the line interface may go to the next descriptor. For each queue stored in the SRAM 154 there is a head and tail address. In this way, the output line interface may tack the queue chain.

First Hit

Generate Collection Print

L6: Entry 9 of 30 File: PGPB Jun 27, 2002

DOCUMENT-IDENTIFIER: US 20020083297 A1
TITLE: Multi-thread packet processor

Abstract Paragraph:
A multi-thread packet processor which processes data packets using a multi-threaded pipelined machine, wherein no instruction depends on a preceding instruction because each instruction in the pipeline is executed for a different thread. The multi-thread packet processor transfers a data packet from a flexible data input buffer to a packet task manager, dispatches the data packet from the packet task manager to a multi-threaded pipelined analysis machine, classifies the data packet in the analysis machine, modifies and forwards the data packet in a packet manipulator. The multi-thread packet processor includes an analysis machine having multiple pipelines, wherein one pipeline is dedicated to directly manipulating individual data bits of a bit field, a packet task manager, a packet manipulator, a global access bus including a master request bus and a slave request bus separated from each other and pipelined, an external memory engine, and a hash engine.

Application Filing Date:
20001222

Summary of Invention Paragraph:
[0007] Methods and apparatuses consistent with the principles of the present invention, as embodied and broadly described herein, provide for a multi-thread packet processor which processes data packets using a multi-threaded pipelined machine, wherein no instruction depends on a preceding instruction because each instruction in the pipeline is executed for a different thread. The multi-thread packet processor transfers a data packet from a flexible data input buffer to a packet task manager, dispatches the data packet from the packet task manager to a multi-threaded pipelined analysis machine, classifies the data packet in the analysis machine, modifies and forwards the data packet in a packet manipulator. The multi-thread packet processor includes an analysis machine having multiple pipelines, wherein one pipeline is dedicated to directly manipulating individual data bits of a bit field, a packet task manager, a packet manipulator, a global access bus including a master request bus and a slave request bus separated from each other and pipelined, an external memory engine, and a hash engine.

Detail Description Paragraph:
[0026] In a packet processor, there is no explicit relationship from one packet to another packet except for the sequence of packets. The packets may be dispatched to multiple processing units or to multiple threads on a pipelined processing engine, as long as the packet sequence is maintained. Because of this, the multi-thread packet processor may be partitioned into multiple packet processing units, each being multi-threaded to keep all execution pipelines filly operating. Since this is a hardware partitioning, the packet sequencing is kept in hardware via PTM 98. As previously mentioned, the multi-thread packet processor may be designed for up to 250 MHz with 4 packet processing units providing 16.5 MPPS with 60 instructions used per packet forwarding decision.

Detail Description Paragraph:
[0048] The state of each thread is independent from the state of all other threads.

Threads and their register content are identified by a Thread Identification (TID) number. Status is provided to indicate which threads are active or inactive, enabled or disabled, etc. In addition to the AM integer pipeline that starts the execution of every AM instruction, each AM has access to several specialized co-processor units such as EMEs 120,266, HE 158, etc. The TID follows the instruction everywhere in the AM or co-processor pipelines. The TID is also the primary mechanism of control between all co-processing units, packet data interfaces, packet pre-classifiers, and the integer pipeline. For most of the interfaces, a TID Queue is used. Each TID queue is b 16.times.4 bit FIFO that contains the thread identifications for some particular operation. Some of the TID queues have multiple write ports to allow new, continued, or co-processor return operations to be started simultaneously.

Detail Description Paragraph:
[0103] To combat this, the multi-thread packet processor allows direct manipulation of bit fields. The problem of bit field isolation, manipulation, and reintegration into the larger data item is handled by the underlying hardware rather than a sequence of instructions as would be done on a general purpose processor The additional hardware increases the processing pipeline depth of each AM, but does not have a detrimental effect on the multi-thread packet processor throughput. For example, consider the problem of incrementing a 5-bit field within a word. The general-purpose processor generally needs to extract the field into a register, increment that register, and insert the field back. For AMs 42,56,70,84, this function is effected using a single instruction:

CLAIMS:

2. The method for processing a plurality of instruction threads according to claim 1, further comprising: transferring data from an input buffer to a packet task manager; dispatching the data from the packet task manager to an analysis machine; classifying the data in the analysis machine; and modifying and forwarding the data in a packet manipulator.

First Hit Fwd Refs

Generate Collection

L6: Entry 13 of 30 File: USPT Mar 9, 2004

DOCUMENT-IDENTIFIER: US 6704794 B1
TITLE: Cell reassembly for packet based networks

Application Filing Date (1):
20000303

Parent Case Text (2):
The following related patent applications are hereby cross-referenced, which are assigned to the same assignee as the present patent application: 1) U.S. patent application Ser. No. 09/914,728, filed Oct. 14, 1999, entitled "Method and Apparatus for Input Rate Regulation Associated With A packet Processing Pipeline" by Prabhas Kejriwal and Chi Fai Ho, 2) U.S. patent application Ser. No. 09/418,683, filed Oct. 14,1999, entitled "Method and Apparatus For Output Rate Regulation And Control Associated With A Packet Pipeline" by Prabhas Kejriwal and Chi Fai Ho, 3) U.S. patent application Ser. No. 09/418,690, filed Oct. 14,1999, entitled "Method and Apparatus For An Output organizer" by Prabhas Kejriwal and Chi Fai Ho.

Drawing Description Text (7):
FIG. 3 shows an embodiment of a control label that is forwarded from the packet aggregation layer of FIG. 2 to the packet processing pipeline of FIG. 2.

Detailed Description Text (7):
With respect to the interface to networking/transport layer 206, note that in the particular embodiment of FIG. 2 the networking transport layer 206 has a packet processing pipeline 240, an output packet organizer 250 and a packet buffer 260.

Detailed Description Text (8):
Other embodiments, however, may implement the approaches discussed herein with other networking/transport layer 206 approaches such as switching architectures (e.g., a crossbar or Banyan switch) or routing architectures (e.g., a general purpose processor coupled to a memory that implements networking/transport layer 206 functionality with software). In the particular approach of FIG. 2, the packet processing pipeline 240 determines how quickly a packet should be passed through system 200, while the output packet organizer 250 holds "packet identifiers" consistent with the determination made by the packet processing pipeline 240.

Detailed Description Text (9):
The type of information processed by packet processing pipeline 240 and entered into output packet organizer 250 is typically control information not random customer data. For example, information located within the various headers associated with a packet (along with other control information as discussed below) is directed to packet processing pipeline 240 from packet aggregation layer 205. Packet aggregation layer 205 is therefore typically designed to extract or copy a packet's header information for presentation to the packet processing pipeline 240.

Detailed Description Text (10):
FIG. 2 symbolically indicates such a scheme by the use of two inputs 270a and 270b. In the embodiment of FIG. 2, packet processing pipeline input 270a carries control information to the packet processing pipeline 240 while packet buffer input 270b

carries a packet's random customer data to the packet buffer 260. The packet buffer input 270b may also be designed to carry the packet header information so that it may be stored along with the packet random customer data.

Detailed Description Text (16):
As discussed with reference to FIGS. 1a and 2, the packet aggregation layer 105, 205 performs cell reassembly in the inbound direction (i.e., toward pipeline 240). Associated with this activity, packet aggregation layer 105, 205 presents (directly or indirectly) control information to the packet processing pipeline 240 and stores packets (with or without its header) into the packet buffer 260.

Detailed Description Text (17):
In an embodiment, once the packet aggregation layer 205 recognizes a complete packet has fully arrived to system 200 and is suitable for further processing by the networking/transport layer 206, the packet aggregation layer 205 forwards control information to the pipeline 240. Up to and before this time, however, the packet aggregation layer 205 continually stores cells (or just the cell payload) that belong to the packet into the packet buffer memory 260. That is, as the packet's cells arrive to system 200 they are stored in the packet buffer 260. Note that in such an embodiment, packet header information is stored along with the packet's random customer data in the packet buffer memory 260 since at least one cell payload will carry packet header information.

Detailed Description Text (18):
After the last cell associated with a particular packet is recognized, the packet aggregation layer 205 forwards the control information associated with the newly arrived packet (which may also be referred to as an input packet) to the pipeline 240. FIG. 3 shows one embodiment of control information 395 that is passed from the packet aggregation layer 205 to the packet processing pipeline 240.

Detailed Description Text (19):
The control information 395, which may also be referred to as a pipeline control label 395 or control label 395, is updated as a packet is effectively processed by the packet processing pipeline 240. Note that in this embodiment, the information 395 includes information 317 located within the various headers associated with the input packet or the cells used to carry the input packet, as well as information 302 that is calculated by the packet aggregation layer 205 or line aggregation layer. This calculated control information 302 may be referred to as a control header.

Detailed Description Text (23):
The length indicator 304 indicates how large the packet is. A user's rate consumption may be measured by the size of the input packet. In order to facilitate a pipeline's rate regulation function(s) (if any), the packet aggregation layer 206 in the embodiment of FIG. 2 presents the packet processing pipeline 240 with an indication 304 of the size of the packet. The corresponding units may vary from embodiment to embodiment. For example, some embodiments may choose to present Length Indicator 304 as a number of cells. Others, again as an example, may choose to present Length Indicator 304 as a number of bytes. In another embodiment it is measured as a number of buffers where a buffer is a collection of cells. Again, the packet aggregation layer is responsible for calculating the length indicator.

Detailed Description Text (32):
If the SOP bit is active, the cell 401a,b, 501a,b at the pipeline input is the first cell of an incoming packet. The SOP bit may be used by the cell processing pipeline 400 to properly identify where a packet's header information starts and where a packet's random customer data begins. For example, consider an exemplary packet where the first 56 bytes are consumed by header information.

Detailed Description Text (35):

Some packet types, such as AAL5 packets, have packet header information located at the trailing end of the packet. The cell processing pipeline 400 may be designed to recognize a trailing packet data/header transition by searching for a flag with the last cell's header or by counting as described above. Associated with parsing is the ability to identify the information within a cell's payload as being either packet header information or the packet's random customer data. Parsing is extracting, copying or otherwise obtaining individual segments of information within the packet header from a cell's payload. Parsing is performed by the parser stage 408 within the cell processing pipeline 400 of FIGS. 4a,b and is described in more detail ahead.

Detailed Description Text (38):
For embodiments having more robust functionality prior to the cell processing pipeline 400 (e.g., SOP is identified prior to the pipeline 400 for all supported packet formats such as AAL5, Frame Relay, Packet Over SONET, etc.), the cell header will have comparatively more information and the pipeline 400 may be devoid of the prior functions. Correspondingly, for embodiments having intermediate functionality prior to the cell processing pipeline 400 (e.g., SOP is identified prior to pipeline 400 for some supported packet formats such as Frame Relay or Packet Over SONET but not other supported packet formats such as AAL5), the pipeline 400 can include the lacking functionality for the packet types that need it. In the cell reassembly embodiment of FIGS. 4a,b and 5a,b, functions that create the cell header 401a, 501a beyond just SOP identification are performed. These include End of Packet (EOP) identification, Packet Type identification and Port Number identification functions.

Detailed Description Text (39):
The EOP bit identifies whether the cell 401a,b, 501a,b is the last cell of an incoming packet. In an embodiment, the EOP bit is used by the cell processing pipeline 400 to indicate that the pipeline control label (395 of FIG. 3) of the packet associated with cell 401a,b, 501a,b should now be sent to the packet processing pipeline 240 of FIG. 2. That is, the EOP bit marks the presence of a fully arrived packet that is ready for processing by networking/transport layer 206 of FIG. 2.

Detailed Description Text (40):
Note that packet processing pipeline 240 of FIG. 2 can be designed to support Layer 2 flows as well as Layer 3 flows. For Layer 2 flows, packet processing pipeline 240 of FIG. 2 regulates and/or controls the delay experienced by a packet within system 200 without reference to information within the IP header of FIG. 3 (e.g., by referring only to Connection ID 310). For Layer 3 flows, cell processing pipeline 400 of FIG. 4 may avoid use of the EOP bit by counting until the length of the packet (as identified by the LENGTH parameter in the IP header of FIG. 3) is reached.

Detailed Description Text (41):
In order to support Layer 2 flows, however, the cell processing pipeline 400 is typically designed to ignore IP header information and therefore should make use of some indication (e.g., the EOP bit) that a packet has fully arrived. In one embodiment, for simplicity, the cell processing pipeline 400 makes use of the is EOP bit regardless if the current packet is associated with a Layer 2 or Layer 3 flow.

Detailed Description Text (42):
Cell header 401a, 501a may also contain the Packet Type parameter. The Packet Type parameter indicates the type of packet associated with the cell 401a,b 501a,b (e.g., AAL5, Packet Over SONET, Frame Relay, etc.). The Packet Type parameter is used by the cell processing pipeline 400 to run various functions that are specific to the type of packet being processed. For example, properly identifying the packet header/data transition within a Frame Relay packet AAL5 packet or Packet Over SONET

packet.

Detailed Description Text (56):
The first stage 413 in, the cell processing pipeline 400 embodiment of FIG. 4b, referred to as the Get Packet State stage 413, obtains information relating to the packet partially transported by cell 401a,b, 501a,b. This information, referred to as packet state information, indicates the extent to which the parsing process has been completed for the particular packet that cell 401a,b, 501a,b belongs to. More details about the contents of the packet state information 406 are discussed further ahead.

Detailed Description Text (67):
In order to store 519a, 519b these data structures, however, the Write Back stage 414 needs to identify a proper memory address for both the Control Label information 411 and the cell payload 401b. In the pipeline 400b embodiment of FIG. 4b, free memory spaces within packet buffer 460 and lookup table 404 are monitored and listed in the free space lists manager 409. The free space lists manager 409 is coupled to the packet buffer 460 and lookup table 404 to fetch, maintain an awareness of and/or control the free memory locations within these devices 460, 404.

Detailed Description Text (74):
The Write Back stage 414 may also check 515 the status of the EOP (End of Packet) value. If the EOP value indicates the cell is the last cell of a packet, the Write Back stage 414 triggers the transmission 516 of the cell's control label (e.g., Control Label 395 of FIG. 3) to the packet processing pipeline. The EOP value, as discussed, may be found in the cell header 401b. For AAL5 packets, the EOP value is found within the cell header of the ATM cell used to carry the packet. For other packets such as Frame Relay or Packet Over SONET packets, the EOP value may be provided by a UTOPIA based interface.

Detailed Description Text (75):
Note that by the time the last cell of a packet is processed by the cell processing pipeline 400, the Control Label 395 of FIG. 3 will be embedded with the cell's packet state information 406 in lookup table 404. That is, since Control Label information 411 identified by the Parser stage 408 is continually fed back to the packet state 406 information within look-up table 404 (as the packet is processed on a cell by cell basis by cell processing pipeline 400), by the time the last cell for a packet arrives, look-up table 404 in many cases has a fully constructed Control Label 395 of FIG. 3.

Detailed Description Text (82):
Control register 610 holds control information associated with the processing of the cell 601a,b. This includes the cell's header 601a. The cell's payload 601b is stored in the cell data register 609. Consistent with the design of pipelined architectures, the control register 610 and cell data register 609 are made available to each stage in the cell processing pipeline (e.g., Get Packet State stage 413, Parser stage 418 and Write Back stage 414 of FIG. 4). That is, these registers 609, 610 correspond to a register (such as register 450a) shown in FIG. 4a. Whether one or more registers is used to hold information pertaining to a single cell is up to the designer.

Detailed Description Text (90):
Connection state information also contains control information for the applicable packet that was calculated by the cell processing pipeline during the processing of the packet's prior cells. This information includes the Packet Identifier 754, Length Indicator 721, Timestamp Insert Flag 722 and Record Route Flag 723. These parameters were discussed with respect to FIG. 3.

Detailed Description Text (110):

Note that the embodiments of FIGS. 4b, 5b, 8 and 9 effectively "copy" the packet header information embedded within a cell payload for storage into the connection state table. As such, all cell payloads are stored in buffer memory including those cell payload portions having header information rather than random customer data. In these embodiments, down stream processing logic (e.g., a packet processing pipeline or a segmentation layer) should be configured to handle packets in buffer memory having header as well as random customer data information. For simplicity use of the term parse or parsing should be construed as covering embodiments that actually parse header information from random customer data (i.e., "extract")or figuratively parse header information from random customer data (such as the copying embodiments of FIGS. 8 and 9).

Detailed Description Text (112):
The Write Back stage may also be designed to selectively direct various parameters within control register 910 depending upon the value of the EOP parameter. Referring briefly back to FIG. 2 as an example, if the EOP parameter is active (indicating that the last cell of the packet is currently being processed by the Write Back stage), the packet has been fully sent to the networking system 200 and is ready for processing by the packet processing pipeline 240 within the networking/transport layer 206. Recall that at this point the control label 395 of FIG. 3 is then forwarded to the packet processing pipeline.

Detailed Description Text (113):
Referring to FIGS. 3 and 9, control register 910 is observed to contain the control label information 395 of FIG. 3. In the particular embodiment of FIG. 9, parsed information 919 corresponds to parsed packet header information 317 (excluding the VPI/VCI) of FIG. 3. The Connection ID 910, The Packet Identifier 954, Length Indicator 921, Timestamp Insert Flag 922 and Record Route Flag 923, along with parsed information 919 are collectively forwarded to the packet processing pipeline.

Detailed Description Text (114):
A selective multiplexor 930 is used to transfer this information (excluding the current state parameter 906) to the packet processing pipeline if EOP is active. If EOP is not active (i.e., the last cell of the packet has not been reached), the selective multiplexor can direct the Packet Identifier 954, Length Indicator 921, Timestamp Insert Flag 922, Record Route Flag 923, Packet State parameter 906 and parsed information 919 back to the connection state memory 620 of FIG. 6. The selective multiplexor 930 may also save the value of the next link pointer 953 of control register 910 into the connection state memory as the link pointer 707 of FIG. 7 in order to properly implement the link list.

Detailed Description Text (117):
Note that in the cell processing pipeline discussed so far a problem may occur if the pipeline simultaneously operates on two cells from the same packet; for example, in a case where two cells from the same packet arrive to the pipeline "back to back". Referring to FIG. 4b, consider a case where a first cell has just been processed by the Parser stage 418, a second cell has just been processed by the Get Packet State stage 413 and both cells are from the same packet.

Detailed Description Text (122):
Note that the discussion so far has pertained to the cell processing pipeline's ability to supply the packet processing pipeline with information that can be used to support Layer 3 and Layer 4 flows. Layer 3 and Layer 4 flows use information within the IP section of the packet header information 317 of FIG. 3 to route a packet through a network. The cell processing pipeline discussed above parses packet header information within a cell payload so that this information can be used by a packet processing pipeline or other networking/transport layer design to route the packet.

Detailed Description Text (128):
FIGS. 11a and 11b compare the operation of the cell processing pipeline for Layer 2 flows and for Layer 3 and 4 flows. FIG. 11a corresponds to the operation of the cell processing pipeline for Layer 2 flows. Note the cell processing pipeline is approximated as only having two stages since the parsing activity of the Parser stage is not performed. Since the Parser stage is only used for simplistic logic operations and next link pointer identification, the cell processing pipeline may be approximated as having only two stages: the Get Packet State stage 1103a and the Write Back stage 1114a. That is, since the speed of a pipeline is determined by the propagation delay of its slowest stage and the Parser stage's activities consume substantially less time than the other two stages, the Parser stage may be approximated as not being involved in Layer 2 flows for analyzing cell processing pipeline performance.

CLAIMS:

29. A method, comprising: a) within a first cycle of a pipeline: parsing a cell payload if said cell payload carries a portion of a packet's header and a portion of said packet's payload; determining packet state information for said packet, where, whether or not a following cell that carries a next portion of said packet carries a portion of said packet's header can be determined from said packet state information; and, b) within a second cycle of said pipeline that immediately follows said first cycle and as a consequence of recognizing that a next cell that immediately follows said cell in the processing sequences of said pipeline is said following cell: using said packet state information to determine whether or not said next cell carries a portion of said packet's header.

52. An apparatus, comprising: a) means for, within a first cycle, parsing a cell payload if said cell payload carries a portion of a packet's header and a portion of said packet's payload; determining packet state information for said packet, where, whether or not a following cell that carries a next portion of said packet carries a portion of said packet's header can be determined from said packet state information; and, b) means for, within a second cycle that immediately follows said first cycle and as a consequence of recognizing that a next cell that immediately follows said cell in a series of pipelined processing sequences is said following cell, using said packet state information to determine whether or not said next cell carries a portion of said packet's header.

63. A networking system, comprising: a packet aggregation layer to perform cell reassembly, said packet aggregation layer comprising: a cell processing pipeline comprising a plurality of pipeline stages, said cell processing pipeline further comprising a pipeline stage to, within a pipeline cycle, 1) parse a payload of a cell if said cell payload carries a portion of a packet's header and a portion of said packet's payload; 2) determine packet state information, where, whether or not a following cell that carries a next portion of said packet carries a portion of said packet's header can be determined from said packet state information; said pipeline stage further comprising, to said parse and to said determine, a micro program sequencer and an execution unit coupled to said micro program sequencer, said pipeline stage coupled to a register, said register to provide said packet state information back to said pipeline stage if a next cell to be evaluated for parsing by said pipeline stage within a next pipeline cycle after said pipeline cycle is also said following cell.